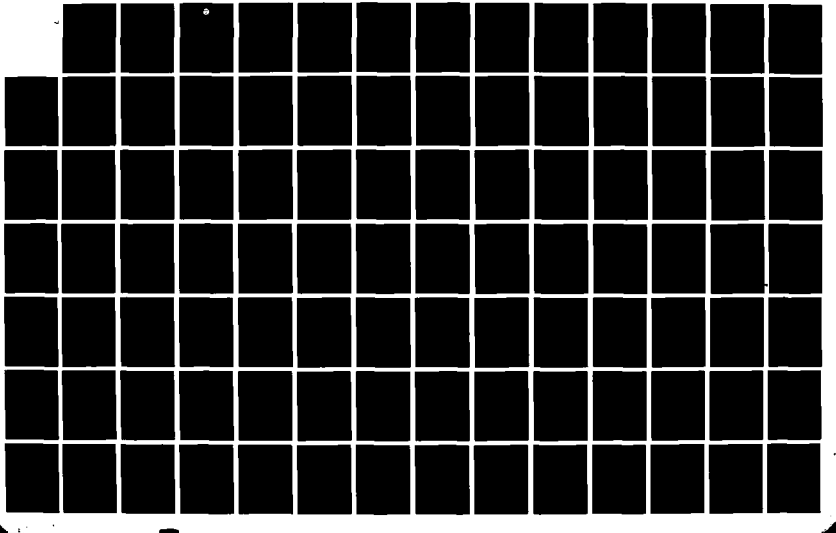
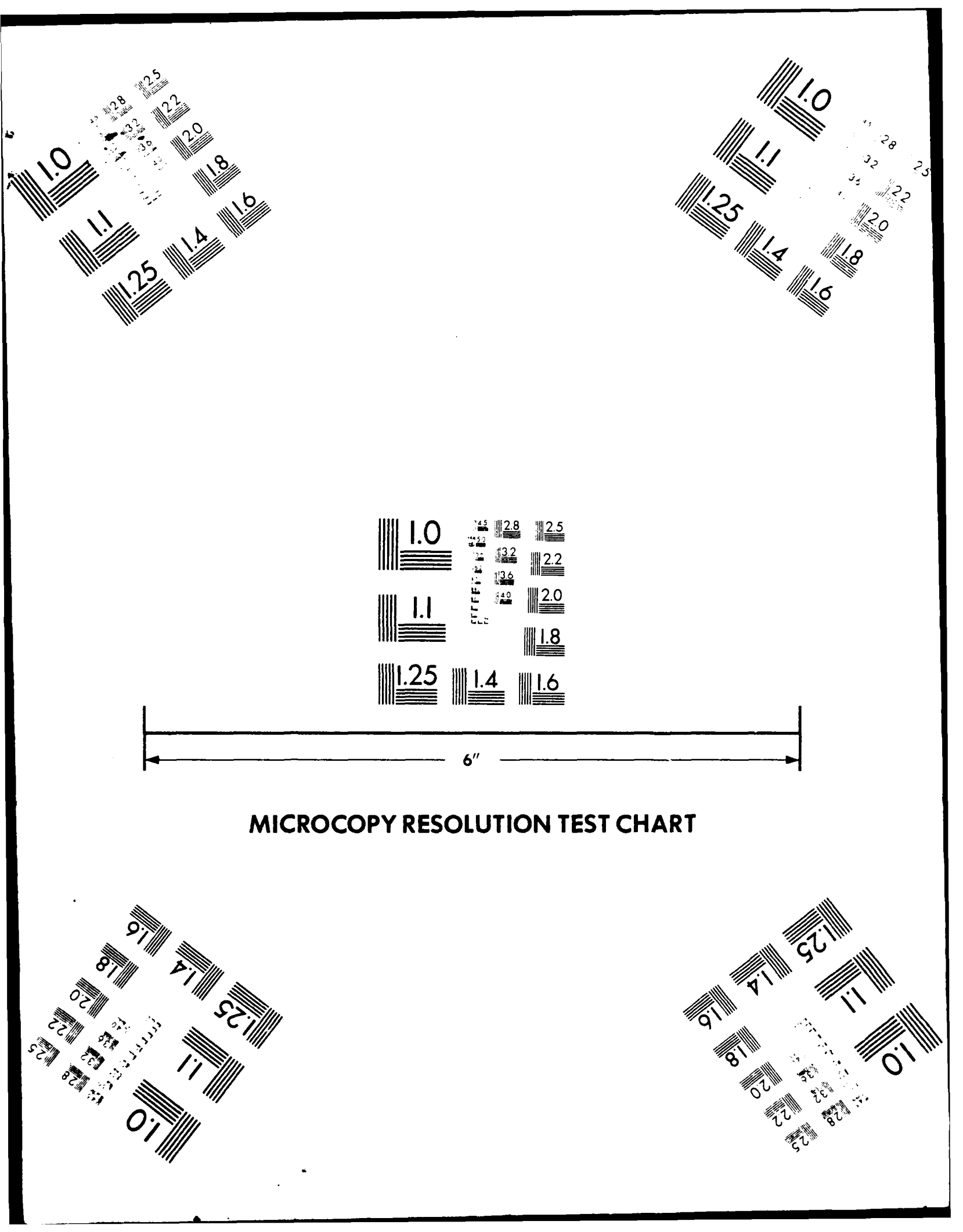
AD-A141868

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA
DESIGN STUDY OF FLOATING POINT SYSTOLIC VLSI CHIP BY JG NASH, GR NUDD HUGHES RESEARCH LABORATORIES

1 OF 2
NOSC CR 232
UNCLASSIFIED
MAR 1984





MICROCOPY RESOLUTION TEST CHART

NOSC CR 232

Contractor Report 232

DESIGN STUDY OF FLOATING POINT SYSTOLIC VLSI CHIP

J. G. Nash and G. R. Nudd
Hughes Research Laboratories

March 1984
Final Report

Prepared for
Naval Ocean Systems Center
Code 8111

Sponsored by
Naval Electronic Systems Command
Code 61R

NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152

NAVAL OCEAN SYSTEMS CENTER SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

J.M. PATTON, CAPT, USN
Commander

R.M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

Administrative information pertaining to Naval Ocean Systems Center Contractor Report 232 is listed below.

Item	Data
Performing Organization	Hughes Research Laboratories 3011 Malibu Canyon Rd. Malibu, CA 90265
Contract Number	N66001-82-M-4120
Controlling Office	Naval Ocean Systems Center Code 8111 San Diego, CA 92152
Sponsor	Naval Electronic Systems Command Code 61R Washington, DC 20360
Program Element	61153N
Project/Subproject	XR02102/01
Work Unit	CG02

The contracting officers' technical representative was Keith Bromley, Code 8111, Naval Ocean Systems Center, San Diego, CA 92152, Tel. (619) 225-7028.

Released by
M. S. Kvigne, Head
Communications Research and
Technology Division

Under authority of
H. F. Wong, Head
Communications Systems
and Technology Department

TJ

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NOSC CR 232	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* DESIGN STUDY OF FLOATING POINT SYSTOLIC VLSI CHIP		5. TYPE OF REPORT & PERIOD COVERED Final 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) J.G. Nash and G.R. Nudd		8. CONTRACT OR GRANT NUMBER(s) N66001-82-M-4120
9. PERFORMING ORGANIZATION NAME AND ADDRESS Hughes Research Laboratories 3011 Malibu Canyon Rd. Malibu, CA 90265 Tel. (213) 456-6411		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N, XR02102/01, CG02
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Ocean Systems Center Code 8111 San Diego, CA 92152		12. REPORT DATE March 1984 13. NUMBER OF PAGES 113
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Naval Electronic Systems Command Code 61R Washington, DC 20360		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Microelectronics
Systolic-array Processing
Multiplier

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The objective of this program was to investigate the feasibility of building a floating point processor (24-bit mantissa and 8-bit exponent) on a single ship based on the Hughes Research Laboratories (HRL) present 28-bit fixed point chip (Multiplication Oriented Processor or MOP chip). The plan was to generate any necessary cell logic, layout, or simulations in order to estimate the size of the chip and predict its performance. Since division and square root were not included in the HRL MOP chip, arithmetic algorithms for performing these operations were to be studied.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

CONTENTS

APPENDICES

FIGURES

TABLES

I. REPORT SYNOPSIS

A. Introduction

The objective of this program was to investigate the feasibility of building a floating point processor (24-bit mantissa and 8-bit exponent) on a single chip based on the Hughes Research Labs (HRL) present 28-bit fixed point chip (multiplication oriented processor or MOP chip)[1]. The plan was to generate any necessary cell logic, layout, or simulations in order to estimate the size of the chip and predict its performance. Since division and square root were not included in the HRL MOP chip, arithmetic algorithms for performing these operations were to be studied.

There were to be at least eight data registers and at least eight serial I/O ports for communication with each of the eight nearest processing element (PE) neighbors. On-chip clocks were desirable and a pin-out arrangement that resulted in a minimum "footprint" ratio was to be minimized. The number representation was to be two's complement, fractional notation, throughout.

Part of our design philosophy is to have all our processor capabilities sufficiently modular so that our chip design can be easily altered to suit the requirements of any desired systolic PE. For example if we added a divider function module, it would have to be bit-slice, carry-save (so that area-time product is $O(n^2)$where n is the bit length), serial/parallel, with all control

---

1. J.G. Nash, S.S. Narayan, and G.R. Nudd, "A VLSI Processor for Adaptive Radar Applications," Proc. 1983 SPIE Conf., San Diego, Aug.21-24 1983.

hardware built-in and capable of running off the special set of high speed clocks provided the multiplier. These considerations, as will be seen later, will influence the choice of algorithms for doing division and square root.

All our designs are based on two sets of two phase non-overlapped clocks. One set operates at more conventional microprocessor type speeds (e.g., 4MHz for the MOP chip) and the other runs approximately 4 to 8 times faster. The high speed clocks are intended for use in serial/parallel type operations such as multiplication, division and serial I/O. In the remainder of the report the two sets of clocks will be referred to as the fast clocks and slow clocks.

The floating point processor described in this report is a "barebones" processor in that it does not support a large number of features that might be desirable in a general purpose processor. For example no capability for various rounding schemes are *included*, *no branching* capability or status flags are provided, and the IEEE floating point standard has not been considered. However, for the primary purpose for which this processor is intended (large systolic arrays), these features would not be of great value. We think it more advantageous to design with throughput considerations given the largest weight.

This report is divided into two sections, the first summarizing the findings of the more detailed second part. In the Appendices we have included some additional relevant material.

I-B. Processor Organization, Size and Performance

In this section we will detail our estimates of the relevant physical information about the chip. Since the MOP chip has already been designed at

5.5 μm drawn feature sizes in NMOS E/D technology, we will give our baseline estimates for this feature size and technology. These numbers should be taken as reasonably accurate. Estimating physical information based on smaller feature sizes depends considerably on the details of the scaling rules and the technology being used. For example the chip area associated with a PE built using a 5 μm polysilicon gate process is not four times the area of the same chip design in the same technology built using a 2.5 μm polysilicon gate process. Many design rules, e.g., registration overlaps, do not scale linearly with gate length. In addition the effective channel length of a MOSFET does not scale linearly with either the drawn or actual gate length.

The basic functional organization of the entire PE is illustrated in Figure 1. There are two system buses connecting any number of "function modules" which are independent units each having their own control sections and in some cases their own fast clock circuitry. Data is sent to and from these units under program control, using the slow clocks.

An approximate functional layout of one version of a floating point processor is shown in Figure 2. The total chip area for 5.5 μm design rules would be 300x343 mil^2, which would come down to approximately 130x149 $mils^2$ for 2 μm drawn feature sizes and 87x100 $mils^2$ for 1 μm drawn feature sizes. The total number of pads would be 19, including 8 control lines, 8 I/O lines, two supply lines and a clock.

The instruction word would be 16 bits, with 8 bits brought in each half clock cycle. There are 8 bits associated with control of each of the two buses shown in Figure 1, 4 bits for a source address and 4 bits for a destination address. Sources and destinations could be registers, adders, I/O ports, etc. All the control associated with complex modules such as the multiplier and

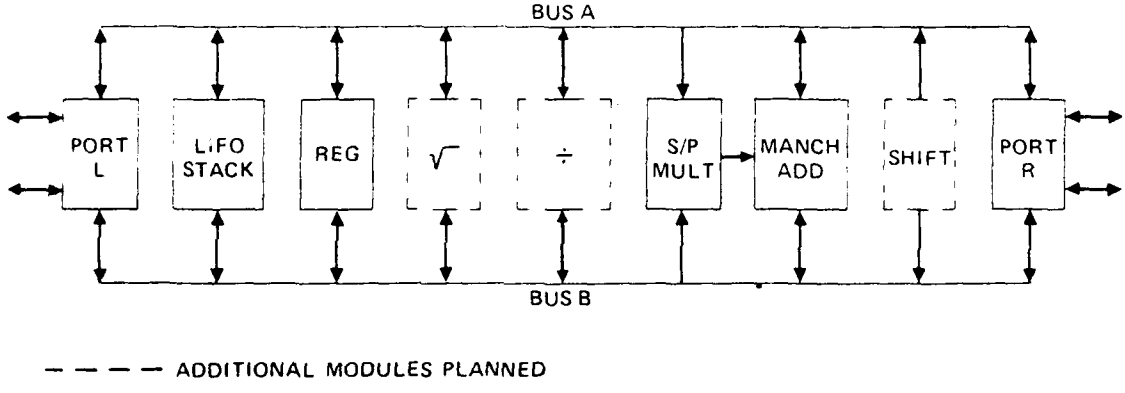


Figure 1. Illustration of processor organization. Function modules are bit-slice and contain all the necessary control and clocking hardware.

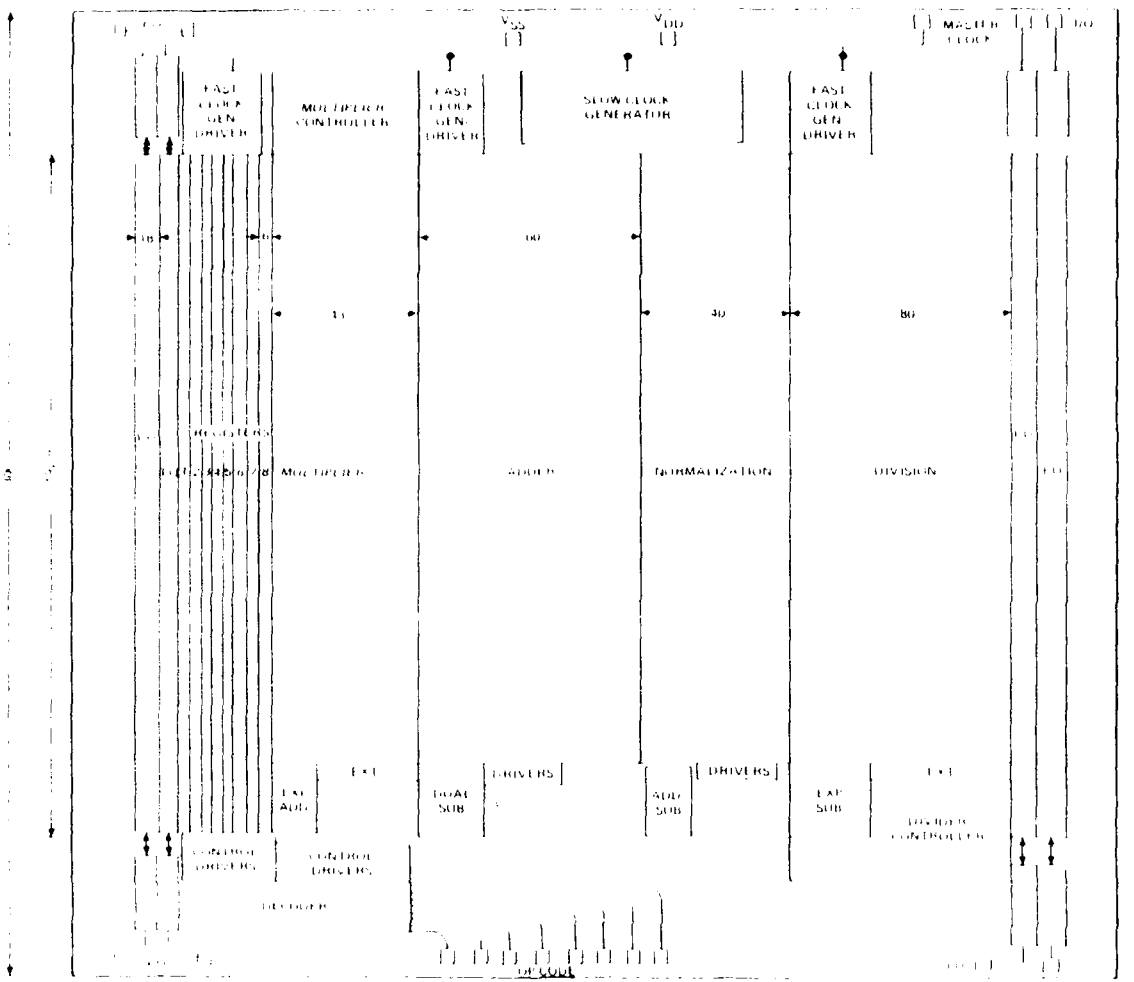


Figure 2 Functional layout of a floating point processor chip with 5.5m design rules. Dimensions are given in mils. For state-of-art design rules (2.0m) expected chip size would be 130 x 149 mils^2.

divider is hardwired into a separate controller that goes with that module. In this way only the control that is necessary is added to the processor.

Since the floating point PE is very modular, there is considerable flexibility in what goes on it. As shown in Figure 2 we have included 8 registers, 4 I/O ports (capable of talking to all eight nearest neighbors), a multiplier, a divider, an adder, and a normalization unit. (The purpose of the normalization modules is to take an operand and scale it so that the most significant bit is always just to the right of the decimal point.) The approximate sizes of each of these is shown so that estimates can be made of PE sizes for alternative configurations.

The expected relative speeds of the different operations (NMOS, E/D) are shown in Table 1. In terms of the number of slow clock cycles a floating point addition will take two clock cycles, one for addition and one for result normalization. A register to register floating point addition will take three clock cycles, one cycle to transfer from a register and add, one for normalization, and one to transfer data back to a register. In terms of absolute performance, present day 2-3 μm feature sizes should provide multiplication speeds of 400nsec and division speeds between 400 and 1300 nsec. By simply putting three multipliers or dividers on a chip and running them concurrently, the effective multiply speeds should be 250 nsec and division speeds between 250 and 900 nsec. A VHSIC technology would provide further improvements in speeds.

Table 1. Performance of proposed floating point chip as a function of nominal and state-of-art design rules.

Function	Absolute Speed (nsec) 5.5μm features	Absolute Speed (nsec) 2 - 3μm Feature Sizes	Concurrent Mode (nsec) 2 - 3μm Feature Sizes
Addition	250	100	---
Normalization	250	100	---
Multiplication	1250	400	250
Division	1250 - 3000	400 - 1300	250 - 900
REG-REG	250	100	---
SERIAL I/O	2500	800	---

It is difficult to estimate power consumption without specific design information, in particular how much dynamic logic is used instead of power consuming static logic. We would expect that it wouldn't be excessive because the MOP chip with almost as many devices (15,000) used only 0.75 W. However, for systolic arrays with several PE's per chip one would have to use a dynamic NMOS logic or CMOS in order to keep the power levels acceptable.

The adder and normalizer have their own shifters (right shifter for adder and left shifter for normalization) so that they can operate independently. The choice of having two separate units for the floating point adder was made for several reasons. First, it would support pipelining. For example if several multipliers were placed on a PE, heavy usage of the adder and normalizer would result in order to add partial sums and carries. By

Table 2. Breakdown of cells that might be used on floating point processor in terms of number of devices associated with mantissa, exponent and peripheral (fixed) circuitry.

FUNCTION MODULE		NO. DEV/MANTISSA BIT	NO. DEV/EXP BIT	FIXED
REGISTER		9	9	50 - CONTROL LINE DRIVERS
LIFO STACK	STACK	8	8	25 (CONTROLLER)
	INTERFACE	6	6	
MULTIPLIER		83	42	100 - CONTROLLER 24 - CONTROL DRIVERS 134 - CLOCK CIR
ADDER	FULL ADDER	56	42	10 - CARRY IN
	EXP COMPARE		30	50 - CONTROL DRIVERS
	SHIFTER	$2\sum_{i=1}^{\ell n(n)} \frac{\ell n(n-i+1)}{2^i} + 13\,\ell n(n)$		
NORMALIZATION CIRCUIT	LEADING 0's COUNTER	19		50 CONTROL LINE DRIVERS
	ENCODER	12		
	SHIFTER	SAME AS ADDER		
	I/O LATCHES	12	12	
	EXP ADJUST		30	
SERIAL I/O		26	26	80 - I/O DRIVERS 57 - CONTROL LINE DRIVERS 134 - CLOCK CIR
PARALLEL I/O		50	50	
CONTROL DECODER				~ 250
SLOW CLOCKS				~ 50
FAST/SLOW CLOCK SYNCHRONIZATION				100
DIVIDER ESTIMATE 2 x COMPLEXITY OF MULT		166	84	500

simultaneously running the adder and normalizer, overall throughput could be increased. This arrangement also allows one the choice of not normalizing results for increased dynamic range or so that loss of significance in operands can be followed, or finally so that throughput could be increased. There is also an advantage in the ease of design and modularity of separate units. In any case the amount of duplicated circuitry (shifter) does not appear to be excessive when compared with the other circuitry.

Each function module that uses a fast clock has its own clocking and clock control unit. In this way one adds no more clock/control circuitry than is necessary for the function modules on the PE. In addition this minimizes the possibility of any skewing problems associated with clock distribution to different chips. For example if the same function modules on different chips receive the master clock slightly out of phase with respect to each other, this will cause no problem because the associated control circuitry will locally count the appropriate number of cycles and then stop the function module. If one of these function modules stops at a slightly different time compared to the other, this small difference will appear negligible compared with the cycle time of the slow clocks which read out the data.

In Table 2 the various cells available for use in the processor chip have been broken down in terms of the number of devices associated with each mantissa or exponent bit, plus a fixed number of devices. The fixed number, which doesn't change for different word sizes, might be the number of devices in a hardwired controller, as in the multiplier, or the number of devices associated with various control line drivers. Since we did not have a specific circuit implementation for the divider (only functional block diagrams), we estimated the device count to be twice that of the multiplier. A summary of

the device count for the PE shown in Figure 2 is as follows:

Floating Point Example

	# Devices
8 Registers	2704
1 Multiplier	2585
1 Divider	5172
4 Serial I/O Ports	4412
1 Adder	2148
1 Normalization Circuit	1674
1 Clock Generator	50
1 Decoder	250
Total	18,995

Thus, the complexity of this minimal floating point processor would be approximately 20,000 devices.

I-C. Algorithms

In this section we briefly discuss our studies of division and square root algorithms. Several basic algorithms for division were investigated:

Direct Methods

Multiplicative Normalization

Combined Multiplicative Normalization and Direct Methods

Iterations Based on Newton-Raphson Formula

CORDIC Algorithm

Our goal was to find an algorithm that would produce an area-time efficient divider that could be easily integrated into our bus oriented, bit-slice, serial/parallel chip framework, with speeds approaching that of our present multiplier. Of the algorithms listed above the direct method is most suited to such a hardware implementation; however, we estimate that it would run four times slower than our multiplier. We feel that division speed can be increased by the combined multiplicative normalization and direct method approach (with the inevitable penalty of increased hardware)*. Although we have not reduced this algorithm to a detailed hardware level implementation, we feel that it offers the possibility of speeds comparable to that of the multiplier. A brief discussion of each algorithm follows.

Direct Method

The direct methods operate in a manner similar to pencil and paper division, by repeatedly subtracting the divisor from the partial remainder (the first partial remainder is the dividend). In each position the quotient is increased by one for each successful subtraction that does not produce a negative result. If a negative result is obtained in some position, the partial remainder is "restored" by adding back the divisor. The divisor is then shifted with respect to the dividend and the subtraction process is begun again.

The direct method is very easy in a binary radix (radix-2) because the number of successful subtractions between shifts can be at most one. Thus, after a shift, if a subtraction is successful, the next subtraction is

---

* We are indebted to Milos Ercegovac for suggesting this algorithm. His analyses are summarized in Appendix A.

guaranteed not to be, so one can shift immediately. If a subtraction fails, rather than restoring the partial remainder by adding back the divisor, it is only necessary to add one-half of the divisor (shift and add) in that position. This division procedure is called nonrestoring division.

There are two basic ways to speed up the non-restoring direct method. The first is to reduce the subtraction time. By representing the quotient digits as redundant numbers it isn't necessary to do a full precision subtraction at each step (no carry propagation required). Instead one can use a carry-save approach (as was done in the multiplier) which allows two words to be subtracted in a time corresponding to a few gate delays, independent of the number of bits. Another speed up approach is to reduce the number of subtraction steps by working in a higher radix. The number of steps is reduced by k, where $r=2^k$ is the radix.

The direct method is very well suited to a design that uses relatively little hardware and fits well with our bit-slice, serial/parallel function modules. The major drawback lies in its slow speed compared to the multiplier. While carry-save partial product subtractions can be implemented easily, proceeding to a higher radix is not easy because it becomes increasingly difficult to select a quotient digit and still have a modular, area efficient design. Another speed limiting factor is the inability to pipeline or overlap operations. Unlike multiplication, the quotient digit is not known ahead of time, but must be selected on the basis of the previous partial remainder and divisor. Therefore, it is not possible to break up the selection of quotient digits and carry-save subtraction steps into smaller operations that can be pipelined. A fast radix-8 divider has been successfully built by Hewlett Packard using a direct division method, but it required 35,000 devices on a

large chip.[2]

A radix-2 implementation of the direct method we estimate would be a factor of approximately 4 slower than the multiplication time. One factor of two arises because of the radix-2 operation, instead of the radix-4 operation of the multiplier. The other factor of two is due to the inability to speed up the quotient digit selection and carry-save operation. With a radix-4 implementation we would gain a factor of two in reducing the number of required subtractions, but the selection operation would be more complex, so the net gain would be less than a factor of two.

Multiplicative Normalization (MN)

For division this algorithm relies on successive multiplications by a number to reduce that number to one. For the division $a=y/x$, if by successive multiplications x is reduced to one, the same multiplications applied to y will generate the desired quotient. If the multiplications are of the form $(1+s_k 2^{-k})$, where k is an integer and s_k is a radix-2 digit, then only shifts and adds are required for this algorithm.

In structure this algorithm is very similar to the direct method described above. It does require considerably more hardware because whatever is done to the divisor, the same must also be done to the dividend. The main advantage of this algorithm is that other elementary functions can be evaluated, such as square root, exponentials, and transcendental functions. However, we mention it primarily as in introduction to the algorithm described next.

---

2. Milos D. Ercegovac, "A Survey of Floating-Point Arithmetic Implementations," Proc. 1983 SPIE Conf., San Diego, CA, Aug. 1983.

Combined Multiplicative Normalization and Direct Method

The main problems associated with obtaining the desired speed from the direct division method are the complexity of the quotient selection process and the inability to pipeline or overlap operations. By combining MN and the direct method in an appropriate way, the quotient selection process can be made very simple and limited overlap of operations is possible. We feel that this approach is the most promising of those investigated in producing and area-time efficient divider.

With this algorithm the input operands are both scaled using MN until they fall within certain prescribed bounds. From this point on the direct division approach is used to generate the quotient digits. The advantage of this approach is that the circuitry associated with selecting the quotient digits from the partial remainder remains relatively simple, independent of the radix chosen for the division. The quotient digits are selected by simple truncation of the most significant radix-r digit of the partial remainder.

There is also considerable speed-up possible in the quotient selection process because it is possible to overlap the calculations of the quotient digit and the partial remainder. For the direct division method described above, the partial remainder had to be computed after the quotient digit selection process was completed.

Although we have not yet reduced this algorithm to a binary level implementation, we feel that it is the most attractive of the division alternatives that we have looked at. Neglecting the operand transformation stage, the division recursion is relatively simple and we estimate that this operation would take only approximately one to two multiplication times. With

a radix-8 version of this algorithm even faster speeds might be possible. On the negative side it is still not clear how to efficiently implement a fast operand transformation capability.

Iterations Based on the Newton-Raphson Formula

Iterative schemes are based on the formula

$$x_{i+1}=x_i-f(x_i)/f'(x_i)$$

which, for a well-behaved function f and a good initial value x_0, can be used to evaluate a root of $f(x)=0$. For example to find a reciprocal we let $f(x)=(1/x)-s$, the root being the desired result. Then the formula above becomes

$$x_{i+1}=x_i(2-sx_i)$$

Division is accomplished by finding the reciprocal of the divisor and multiplying by the dividend. The most important feature of this algorithm is that it converges quadratically. For example if a small lookup table is used to find the first four bits of the result, then the first iteration will produce a new result accurate to eight bits, the second iteration 16 bits, and so on. Thus, the convergence rate is O(log n), rather than O(n) for the other algorithms we have looked at, where n is the bit length. Other elementary functions can be evaluated in this way using only multipliers.

The disadvantage of this approach is that it requires a considerable amount of hardware (a look-up table ROM and very high speed multipliers) which isn't easily integrable into our design scheme, and it isn't any faster than

alternative algorithms we are looking at. For example, it is our goal to do divisions at the same rate as multiplication.

CORDIC Algorithm

The primary attraction of the CORDIC algorithm is its generality. If a wide range of elementary function evaluation is desirable, then this is probably the best alternative. Conventional implementations of the algorithm require n time steps, each of which involves a full n precision addition or subtraction and a shift. We think that speed improvements can be made to the algorithm to eliminate all the n precision additions and possibly to eliminate some of the shift requirements. However, in any case the hardware needs of the algorithm are considerably greater than required for the other division algorithms. The basic needs are three adder/subtractors, a small ROM, and two shifters.

Square Root

We have looked at several square root algorithms and generally found the problem similar to that of division. For this reason we concentrated our efforts on examining the division problem. However, we do describe later an algorithm based on the odd series approximation as an example of a direct approach to performing square roots that would support a hardware implementation well suited to our design scheme.

II. Detailed Description of Floating Point Design

II-A. Number Representation

In this section we briefly describe the characteristics of the 32-bit word, although it is intended that the manner in which the chip is organized will enable it to be assembled rapidly with any word length.

For this 32-bit word the mantissa is 24 bits in length including one sign bit. The fractional, 2's complement notation increases efficiency of computation. The range of numbers representable is then $1-2^{-23}$ to $-1+2^{-23}$ ($1-1.1920929 \times 10^{-7}$ to $-1+1.1920929 \times 10^{-7}$).

The exponent is represented as 8 bits in excess 128 notation. In other words the exponents are biased by 128. This simplifies some of the manipulation of exponents because it eliminates negative exponents. The exponent range is then 2^{-128} to 2^{127} (2.94×10^{-39} to 1.701×10^{38}).

When exponents are subtracted as in division the effect of the exponent bias is simply canceled. However, in multiplication the bias is added twice and must be subtracted out. This can be done with a simple circuit that has as inputs the most significant bits (MSB) and carry into this position. For example the exponent addition

$$\begin{array}{l} A_1\ A_2 \cdots A_8 \\ B_1\ B_2 \cdots B_8 \\ \hline C_1\ C_2 \cdots C_8 \end{array}$$

uses the truth table shown in Figure 3(a) to determine the MSB C_1 which can be implemented by the circuitry shown in Figure 3(b).

II-B. Clock Generator and Control Circuitry

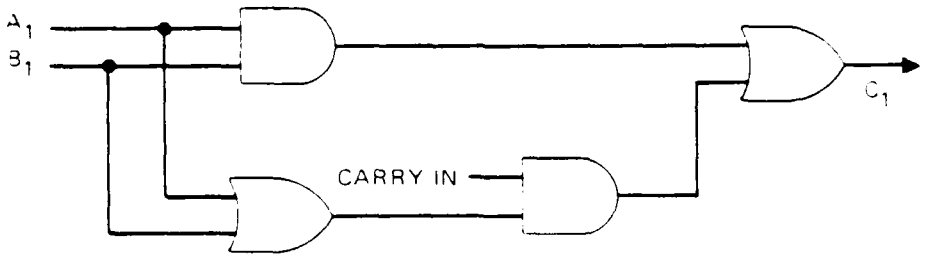
Overall array synchronization will be based on a single phase high speed clock (~16MHz for 5.5 μm design rules or ~50MHz for 2 μm design rules) made available to every PE. Each PE will derive from this a set of low speed, or system clocks, for use in transferring information between function modules and between chips, and a set of high speed clocks to drive arithmetic modules and serial I/O ports. It is expected that the ratio of high to low speed clock frequency will be between 4 and 8. (For the MOP chip it was set at 4.) In this section we describe circuitry intended to perform these functions.

High Speed Clocks

The primary design goal for the high speed clock circuitry is to avoid problems associated with possible skewing in the distribution of the clock signal. This can be accomplished by localized clock generator and control circuitry. Each high speed function module will have a counter circuit that can be "programmed" to perform a certain number of counts of the high speed input clock and then to shut off the local high speed clock drivers. For example our 28 bit fixed point MOP chip multiplier would require the clock controller to count 16 clock cycles and then stop the local clock. With this arrangement, if the high speed clocks in arithmetic modules on different chips are out of phase, they will finish their operation at only small fractions of

A_1	B_1	CARRY IN	C_1
1	1	X	1
0	0	X	0
0	1	1	1
1	0	1	1
1	0	0	0
0	1	0	0

(a)



(b)

Figure 3. (a) Truth table for correction to most significant exponent bit, and
(b) Circuit to perform this logic.

the time associated with the slow clock cycle. From the point of view of the slow clocks, which will read information out of the function modules, these small differences in time will not be important. Thus, high speed clock skewing will not lead to any synchronization problems.

A possible control circuit and partial timing diagram are shown in Figure 4(a) and 4(b) at a functional level. Operation begins with the "Load Function Module" going high, indicating that operands are being loaded into this function module. This signal resets the counter and flip-flop 3 (FF3), which is used later to reset FF1. When the "Load Function Module" goes low, indicating that the operands are loaded, the output of FF1 goes high enabling the high speed clock input via the AND gate. The output of the AND gate drives FF2, whose purpose is to provide clean beginning clock waveforms to the clock drivers and to the counter circuit. When the counter reaches its programmed value it issues a "DONE" signal which FF3 uses to reset FF1. The output of FF1 then goes low, disabling the high speed clock input to the clock driver circuits. The way the circuit is drawn in Figure 4(a) indicates that there will be several gate delays associated with the circuit "shut down" operation. This consists of a few gate delays through the counter, plus single gate delays through FF3, FF1, and the AND gate. This long total delay can be avoided by pipelining the control operation with the addition of a little circuitry. The count gate will have to be set to decrease the "count" by the number of pipelined stages.

A fast synchronous parallel counter design is shown in Figure 5. The multi-input AND gate to each flip-flop is best built using a high speed NOR implementation. The "count gate" is simply an AND gate with counter outputs as inputs. When the counter reaches the desired number this gate is responsible

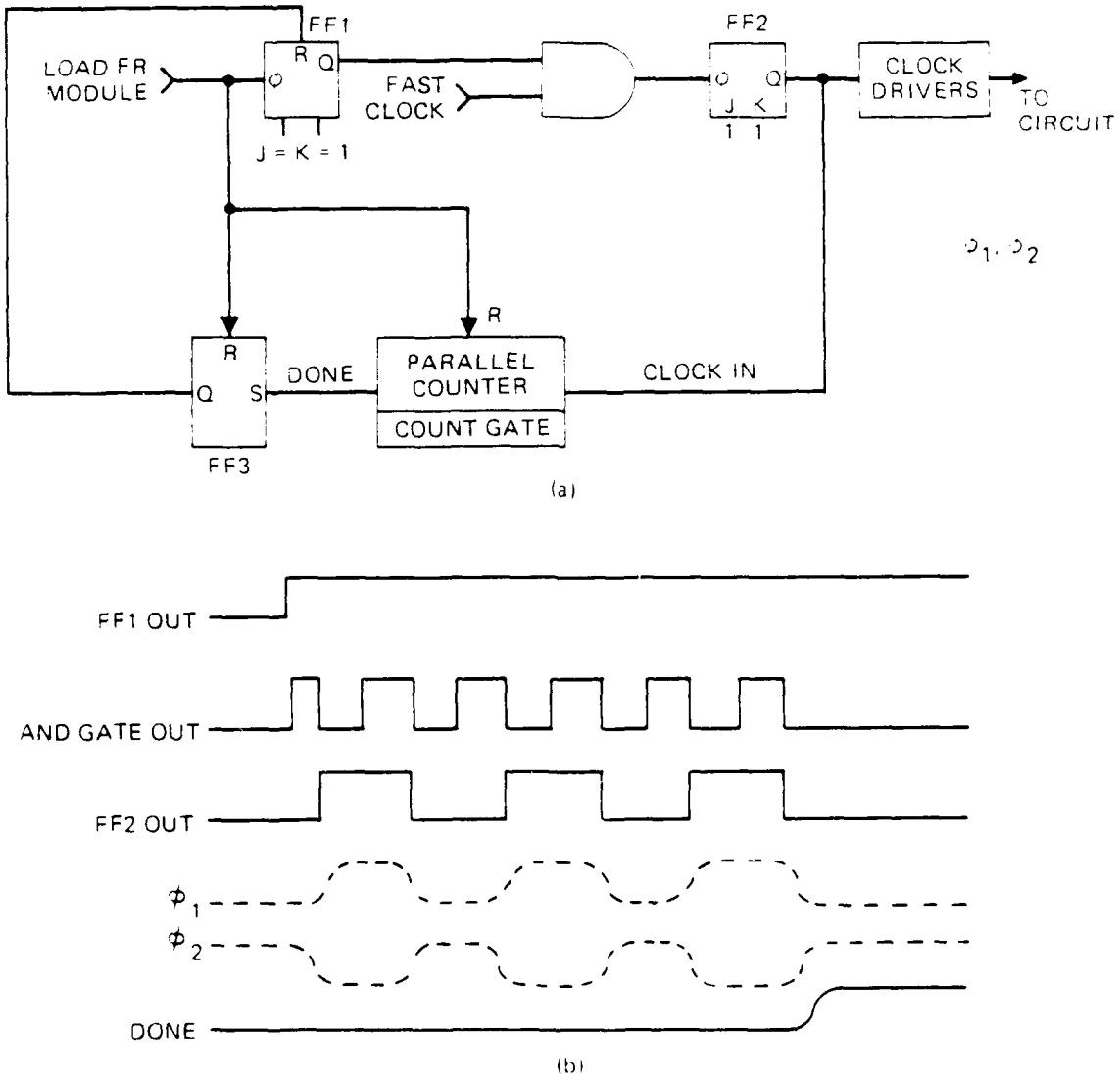


Figure 1. (a) Functional block diagram of high speed clock driver and control circuit, and
(b) Corresponding waveforms.

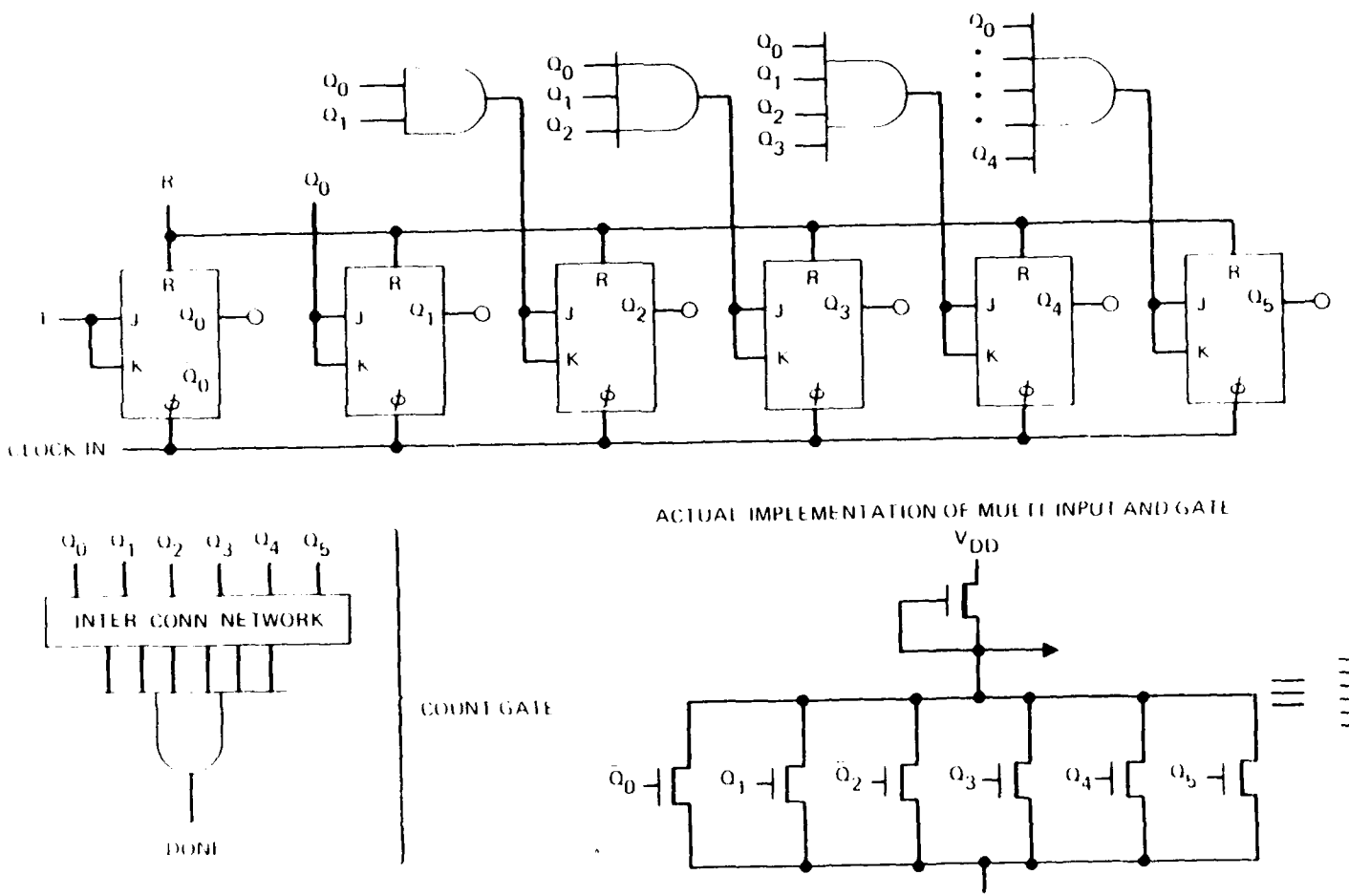


Figure 5. Logic diagram of parallel counter implementation.

for stopping the clock driver operation.

One possible implementation of the J-K FF's seen in previous figures is shown in Figure 6 based on a fast NOR gate implementation. Approximately 27 devices are used per FF.

Clock driver circuitry is shown in Figure 7. This configuration will produce a non-overlapped two phase clock from a single phase input. This type of circuit could possibly be used as a driver for high and low speed clocks because current drive capabilities should be similar.

II-C. Multiplier

Conversion of our fixed point multiplier to floating point basically reduces to the problem of adding an exponent handling unit that doesn't introduce major topological irregularities. A block diagram of our proposed structure is shown in Figure 8. The carry-save Booth's multiplier circuit is described in detail in Appendix B.

The exponent handling section consists of a set of registers to store the data, a simple ripple adder to perform the exponent addition, and an output latch. We can use a minimum device combinatorial full adder cell shown in Figure 9 in order to conserve area. This full adder cell is not as fast as that in the mantissa processing section, but it only has to add two small numbers in approximately four of the slower clock cycles (1 sec for 5.5 m feature sizes). We have laid out a full adder cell and simulated it (including parasitic capacitances) using SPICE to determine its speed characteristics. For 5 m feature sizes the propagation delay through the cell is less than

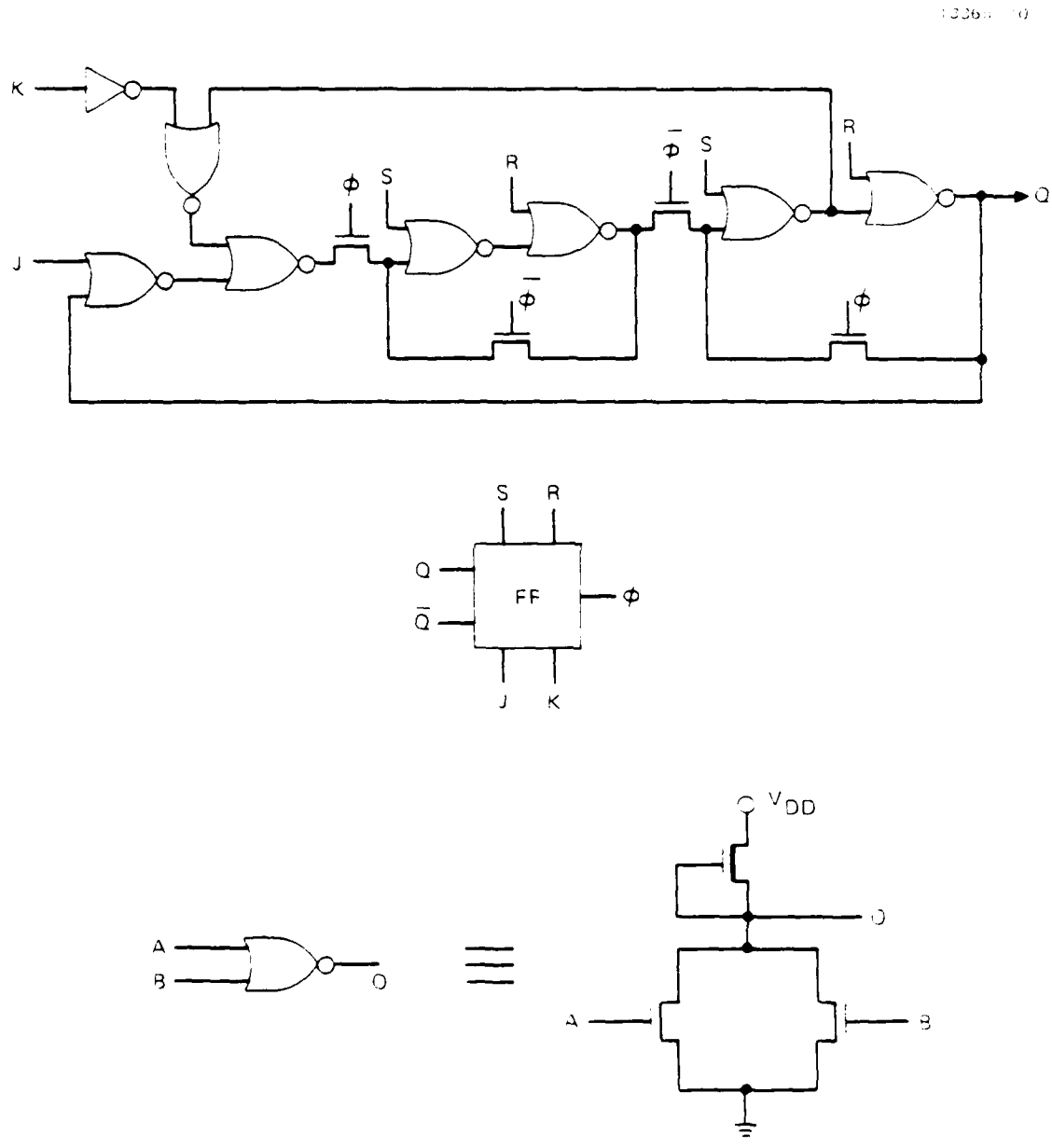


Figure 6. Logic and circuit diagram of JK flip-flop.

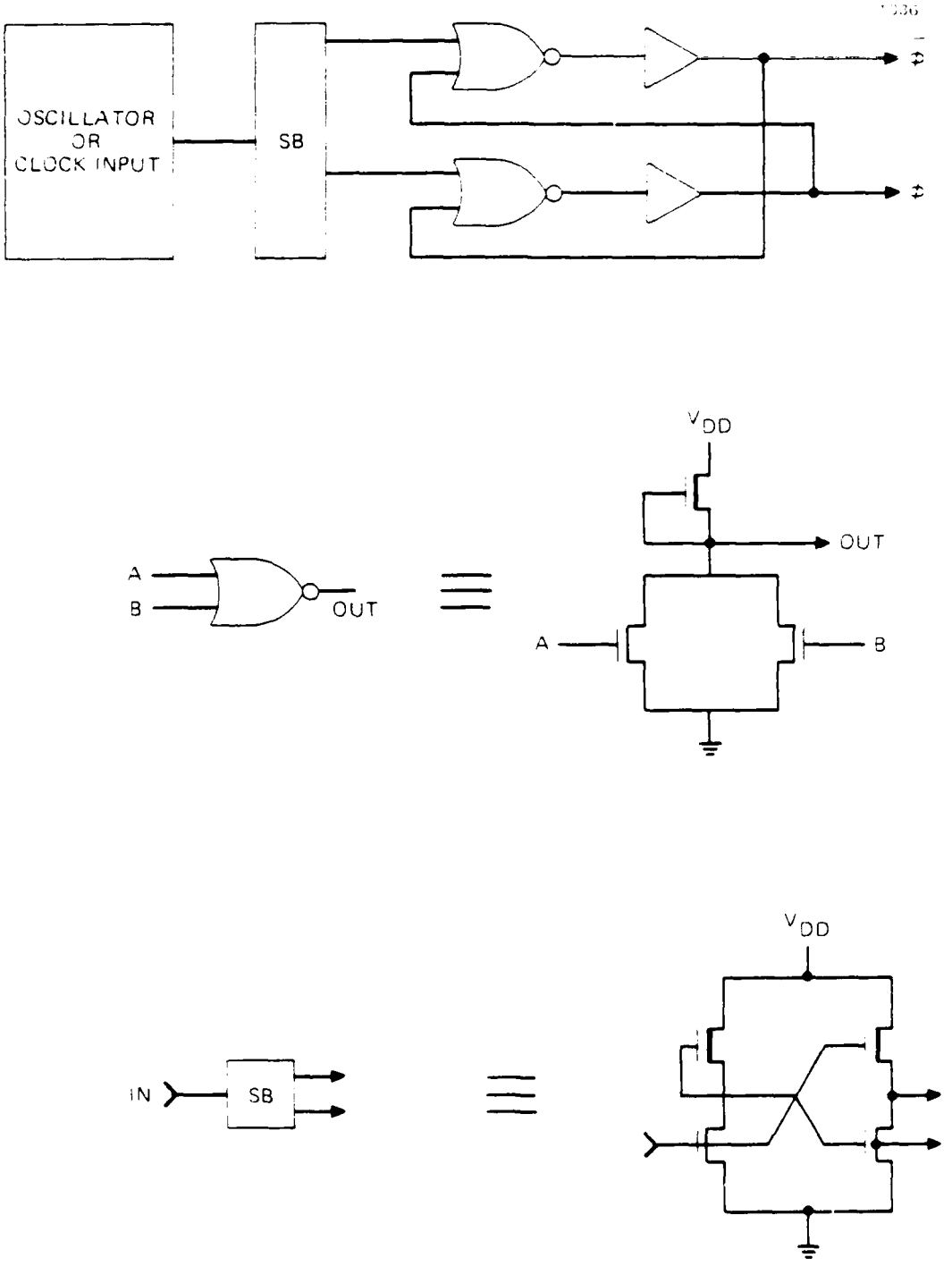


Figure 7. Logic and circuit diagram of clock driver.

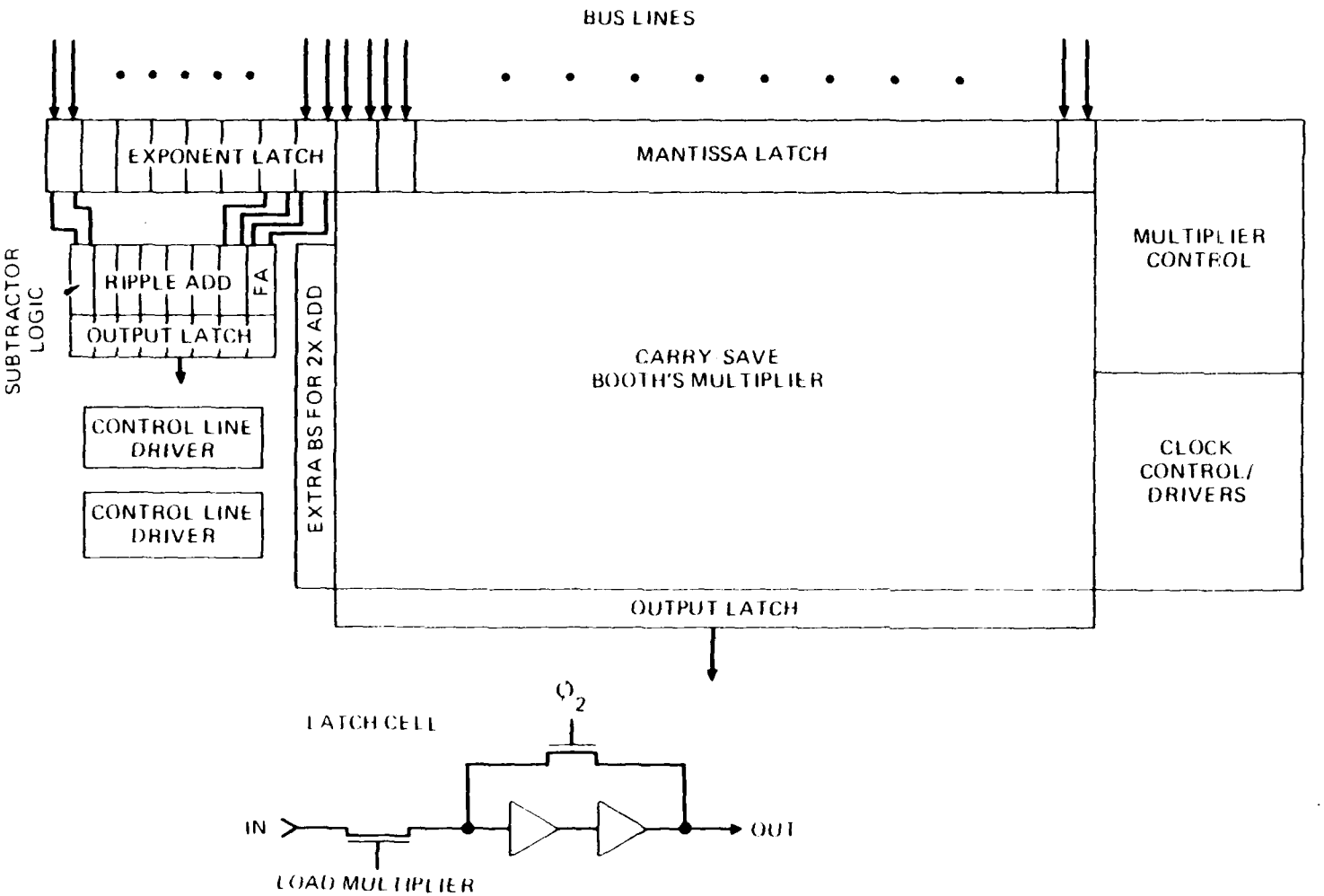


Figure 8. Function layout of floating point multiplier.

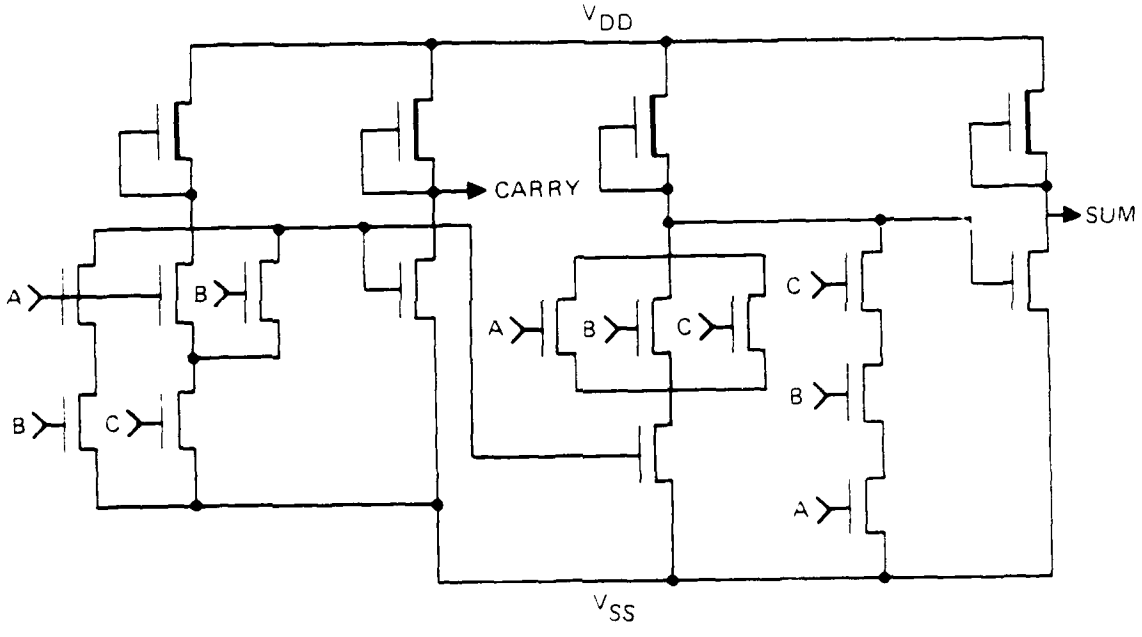


Figure 9. Minimum device full adder circuit, with inputs A, B, and C and outputs sum and carry.

100nsec, or an acceptable 800nsec for an 8-bit exponent.

As discussed in Section II-A the full adder associated with the most significant exponent bit will have some extra logic in it to generate the correct results for the 128 bias.

Some experimentation will be necessary to determine an optimal layout for the exponent section. Possibly control line drivers will fit well into the available area as shown in Figure 8 or it might be possible to place the exponent ripple adder vertically.

The operation of the multiplier is no different than for the fixed point version. The operands are loaded simultaneously into the input latches and then after four slow clock cycles (for MOP chip) a sum and a carry result will be available in the output latches. This result must then be sent to the adder to propagate the carry. The exponent section is purely combinatorial, requiring no special clocks.

II-D. Adder

The fixed point adder requires considerably more added to it than the multiplier in order to create a floating point capability. There are two basic operations required to perform an addition. First, the mantissas are aligned and added. Then this result is normalized in some way. Since each of these operations is difficult to perform, we decided to split these operations into two sets of hardware. This simplifies the control circuitry and increases speed since both units can operate simultaneously. An added feature is that the programmer has the option not to normalize his results. This can be of

advantage to someone who needs increased dynamic range (up to 2^{23}) or who needs to follow closely the loss of significance in his arithmetic operations.

The entire process of addition and normalization, as described below, is expected to take two clock cycles, or three cycles for a register to register operation. One clock cycle is associated with transfer of the operands from another function module followed by addition, one clock cycle for normalization, and one to return the result to another function module.

For subtraction operations we note that it is only necessary to take the two's complement the appropriate operand, introduce an appropriate carry into the least significant bit position, and then add. The complementation of the operand is expected to be done ahead of time. This is taken care of most easily by a feature of the adder output that allows its result or its complement to be placed on the bus. If it is known ahead of time that a result will be subtracted later, then the output complement is selected.

The remainder of this section is divided into two parts, that on the addition and that on normalization of the result.

Addition

The basic problem in addition is alignment of the mantissa. This is normally done in three steps: determination of the larger operand, shifting the smaller operand mantissa, and adding the results. The corresponding hardware to perform each of these functions is shown in Figure 10. The circuit consists of two input latches to hold the operands, a subtractor to determine the larger of the two, a shifter to align the mantissa and a conventional ripple type adder.

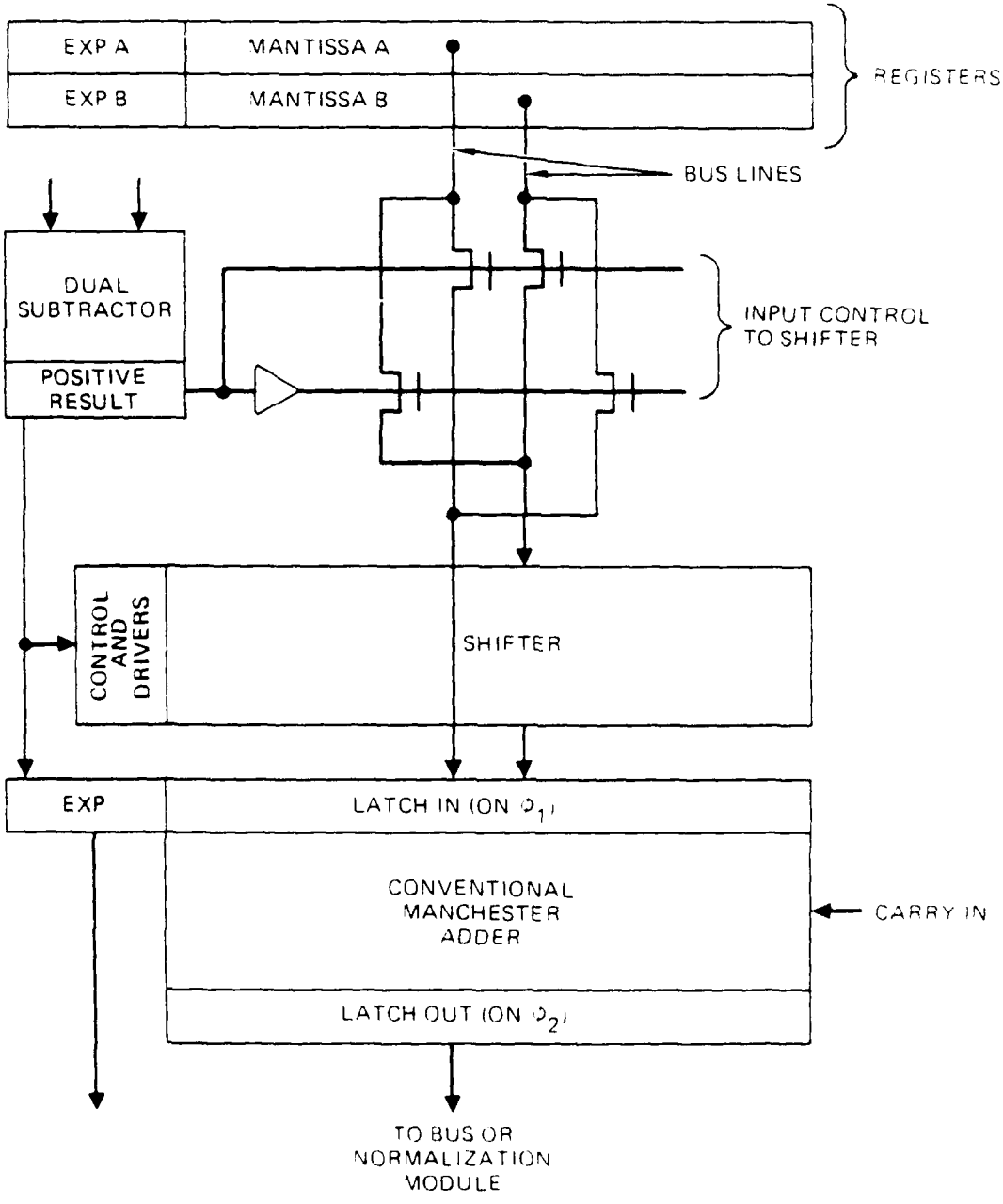
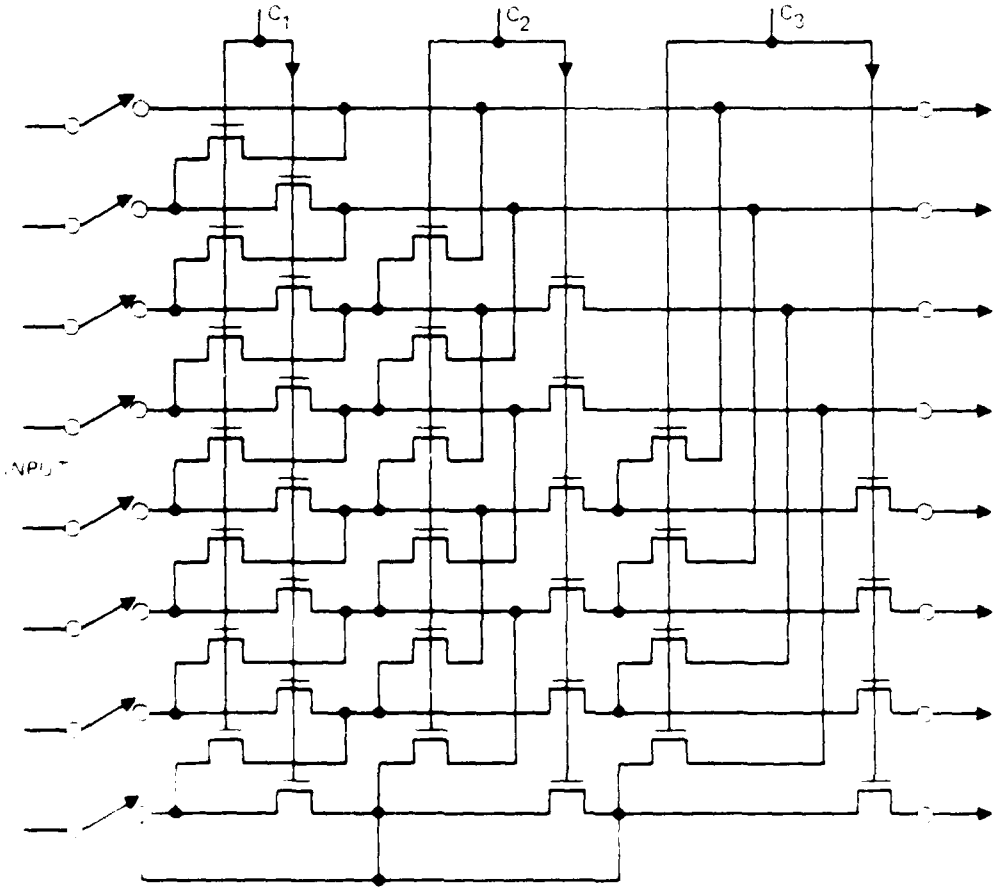


Figure 10. Functional layout of floating point adder.

The first half of the first clock cycle is used to put the operands on the bus (from a register or other function module) and shift the mantissa of the smaller operand. During the second half of the first clock cycle, the two operands are added. The second clock cycle could be used to return the result to a register (or to the normalization circuit). The time associated with driving the buses is fairly small because it is a pull-down operation. Spice simulations we have performed (Appendix C) show that for 5 μm feature sizes a 2pf bus line (approximate capacitance of that on MOP chip) can be pulled down with a transistor having W/L = 6 in approximately 30nsec. This leaves 95nsec (for 5.5 μm, 4MHz slow clocks) to perform the subtraction of exponents and shifting. If a Manchester type subtractor is use for the exponent section, approximately 36nsec will be required to do the subtraction, leaving an adequate 59nsec for shifting and error margin. (We assume that for smaller feature sizes these times will scale proportionately.)

Since one doesn't know ahead of time which of the exponents is larger, the output of the subtractor can be either positive or negative. Although this information is sufficient to determine which operand is larger and can be used to generate the control signals to the pass transistors controlling the input to the shifter in Figure 10, it is not sufficient to control the shifter itself, shown in Figure 11. The shifter needs at its control inputs (C_i in Figure 11) binary values corresponding to the number of bits of shifting desired. For this reason it is necessary to build the dual subtractor unit shown in Figure 10. This unit will produce two outputs, corresponding to the two possible subtractions of the operands. The positive output will always be used to control the shifter. It would be possible to use the same subtractor to perform both subtractions, but the time lost would probably result in the addition of an extra slow clock cycle to the overall addition time. The



CONTROL SIGNALS			OPERATION
C_1	C_2	C_3	
0	0	0	NO SHIFT
0	0	1	SHIFT 1
0	1	0	SHIFT 2
0	1	1	SHIFT 3
1	0	0	SHIFT 4
1	0	1	SHIFT 5
1	1	0	SHIFT 6
1	1	1	SHIFT 7

increased hardware is small compared to the total dedicated to the entire addition operation.

The shifter network in Figure 11 consists of a set of bus lines connected by pass transistors. The set of pass transistors controlled by C_1 shifts bits by one, those controlled by C_2 shifts bits by two, and so forth in a binary progression. The maximum number of pass transistors in series will then be of order log(n) (e.g., 4 for a 24 bit mantissa). We expect then that the shifter will not consume much area (less than half that of the ripple adder). The delay through four pass transistors should be equivalent to one gate, therefore making it very fast.

Sign extension is a very important consideration for two's complement arithmetic. As a word is shifted, the bits which are shifted over must be set equal to the sign bit. This feature is introduced by connecting the sign bit to the pass transistor inputs at the bottom of the shifter array as shown in Figure 11.

After the operands pass through the shifter they are latched at the end of the phase one half of the slow clock cycle into the ripple adder (Manchester type adder). During the phase two of the slow clocks, the mantissas are added. The results are available to be transferred to another function module on the next clock cycle.

If both of the operands have the same sign, it is possible that there could be overflow during the addition. For this reason there are 25 full adder cells instead of 24. All 25 bits can then be passed on to the normalization unit to perform the right shift of data by one bit. If the output of the adder is to go to a function module other than the normalization unit, only the most

significant 24 bits are put on the bus.

Normalization

The problem of normalizing a word or shifting it appropriately until the most significant bit is just to the right of the decimal point is actually more difficult than the floating point addition described above. The basic functional blocks, shown in Figure 12, are similar to those of the adder section. There are two possible inputs, one directly from the adder, and the other from one of the chip buses. As mentioned above there are 25 inputs from the adder and only 24 from the chip buses.

The functional block labeled "Significant Bit Counter" counts the number of leading "1's" or "0's" (nonsignificant bits) using a circuit such as that shown in Figure 13. The first logic stage in Figure 13, consisting of exclusive OR gates, is used to identify changes in data polarity from bit to bit. The NOR chain uses this information to generate an output in which the number of ones is equal to the number of leading "1's" or "0's". The last set of exclusive OR gates detects the position of the 1 to 0 transition, which marks the position of the desired decimal point.

The encoder section takes the above described output and generates the binary equivalent of the number of shifts required to correctly normalize the input. This circuit deviates slightly from the concept of identical bit-slice elements in that each encoder slice must generate a binary number equivalent to its position in the word. However, as can be seen, each encoder slice is built identically except for five very short connections which are used to set the binary count. The binary count is either added to or subtracted from the exponent depending whether the shift is to the right or left, respectively.

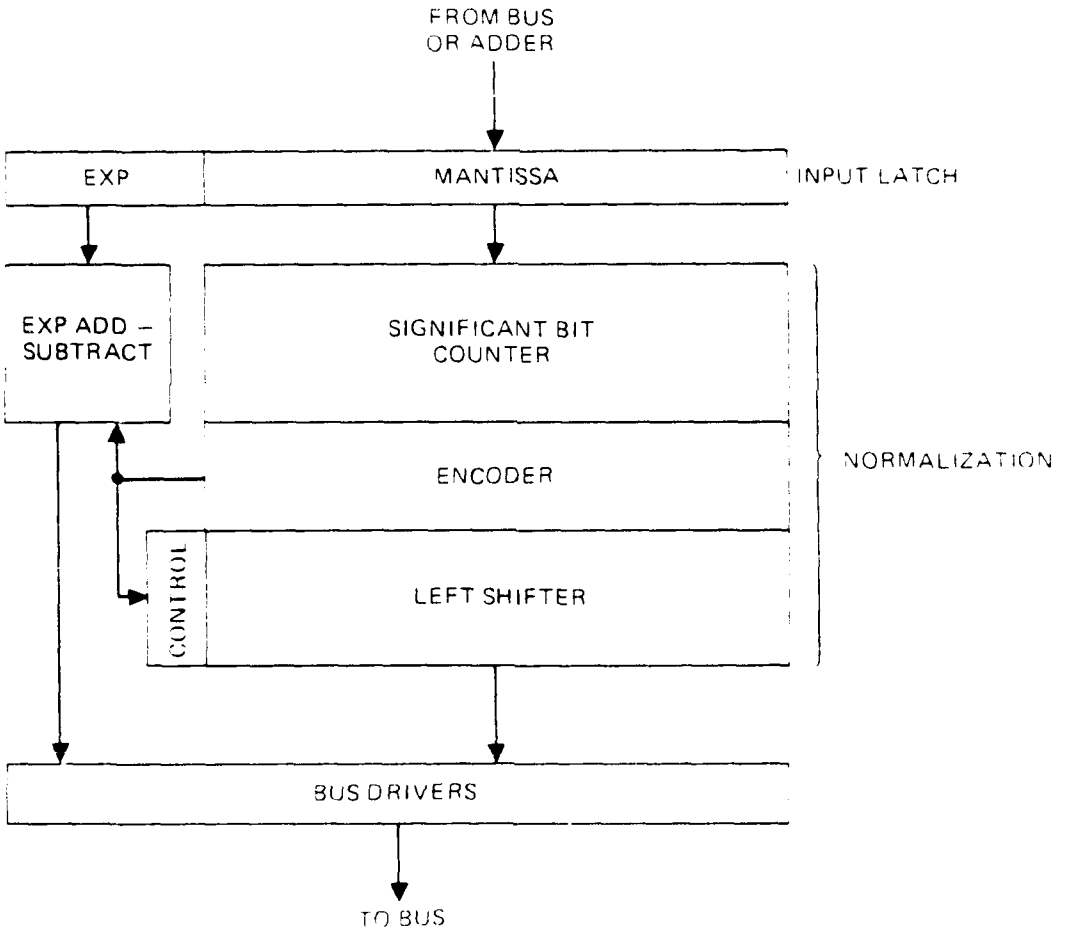


Figure 12. Functional layout of normalization circuit. Actual shift operation takes place at the beginning of the slow phase one clock cycle.

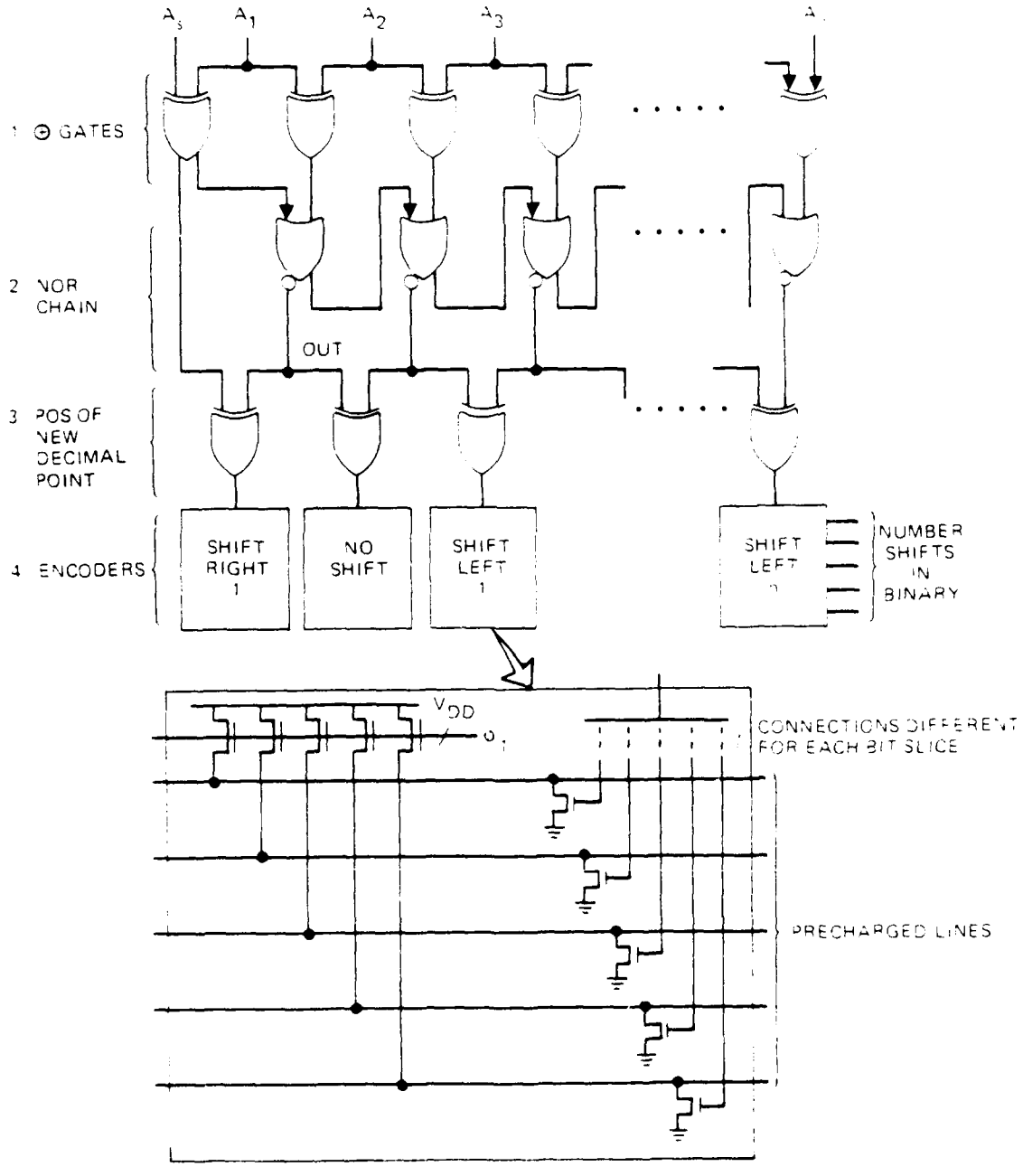


Figure 13. Circuitry used in the various sections shown in normalization circuit shown in Figure 12. One bit-slice cell of the encoder is shown expanded.

The shifter is identical to that in Figure 11 except that shifts are in the opposite direction (left) for normalization. As a result the shifter hardware will be identical to that in Figure 11 except that it is reflected about its shorter axis.

The operation of the circuit begins with the latching of the input word from the bus or adder at the end of slow clock phase one. (The encoder outputs have been precharged during slow clock phase two.) After loading the input word, the encoder outputs are latched to the exponent adder/subtractor at the end of phase two. As in the adder, exponent handling and shifting are done during the phase one clock cycle.

The primary time bottleneck is in the NOR chain which counts the number of nonsignificant bits. If this NOR chain is too slow, circuits are available that can perform the same function using a carry-propagate approach as in the adder section.

II-E. Serial I/O Ports

Communication requirements between adjacent PEs are an important consideration in systolic array design, due to the large amount of information passed and the large number of PEs on the receiving end (as many as eight). In order that there be a balance of communication and computational needs we propose adding at least four bidirectional serial I/O ports to each chip, organized as shown in Figure 14. As can be seen, the I/O ports are arranged in a natural way to aid flow of data through PEs. If higher bandwidths were required each I/O port could be replicated the appropriate number of times. For example, with eight I/O ports two words could be passed simultaneously in a

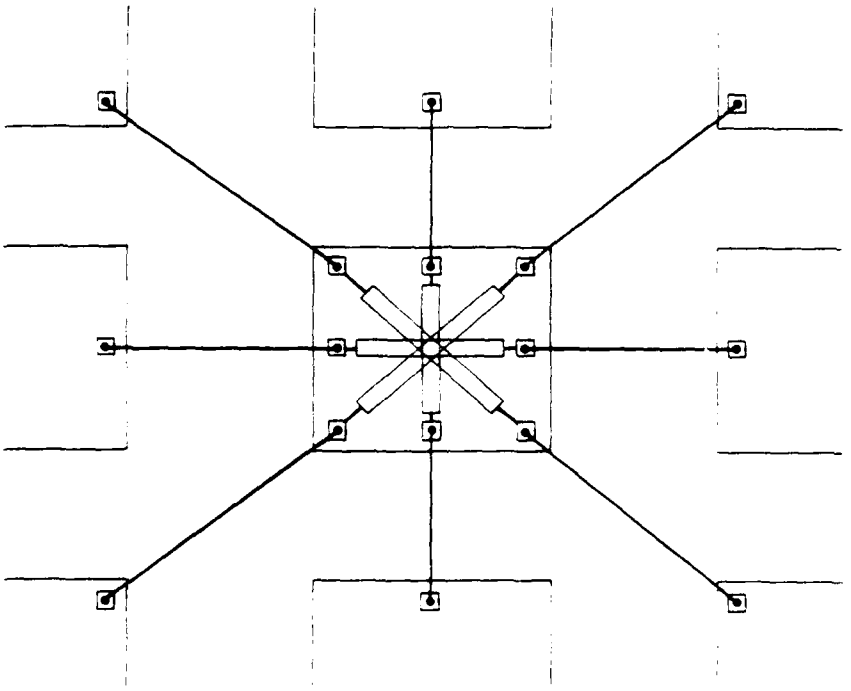


Figure 14. Proposed serial I/O port organization.

given direction between any two PEs.

A functional block diagram illustrating the major components of a serial I/O port is shown in Figure 15(a). In order to facilitate integration into a bus oriented processor the I/O port is built as a dual port register-shift register combination (parallel/serial multiplexer). The register is capable of reading or writing to either of two buses using the read A or B, write A or B control lines.

In order to increase I/O transfer rates it is natural to use the high speed clocks available to each PE. The high speed clock control circuitry, shown in Figure 15(b), is identical to that described in Section II-B. These clocks are supplied to the I/O function module along with a control signal M, which determines the direction of the shift and also disables the appropriate driver circuitry as shown. A major consideration in the serial I/O design is the loading requirements. If each PE occupies a single chip, the output load would be at least the capacitance associated with a couple of pads. In order to match this drive requirement with that of the shift register stages, pipelined output driver stages need to be added. The overall transfer speed would suffer only slightly due to the increased latency if the number of pipelined stages were much less than the word length. These stages would be modular and could be added as necessary. An example of one pipelined stage is shown in Figure 16.

The operation of the serial I/O function module would be very straightforward. The operation as a register would require the same control signals as any other register. An error could occur only if one tried to write or read from a register while it was in the process of transferring information. To send data to an adjacent PE one would have to select the

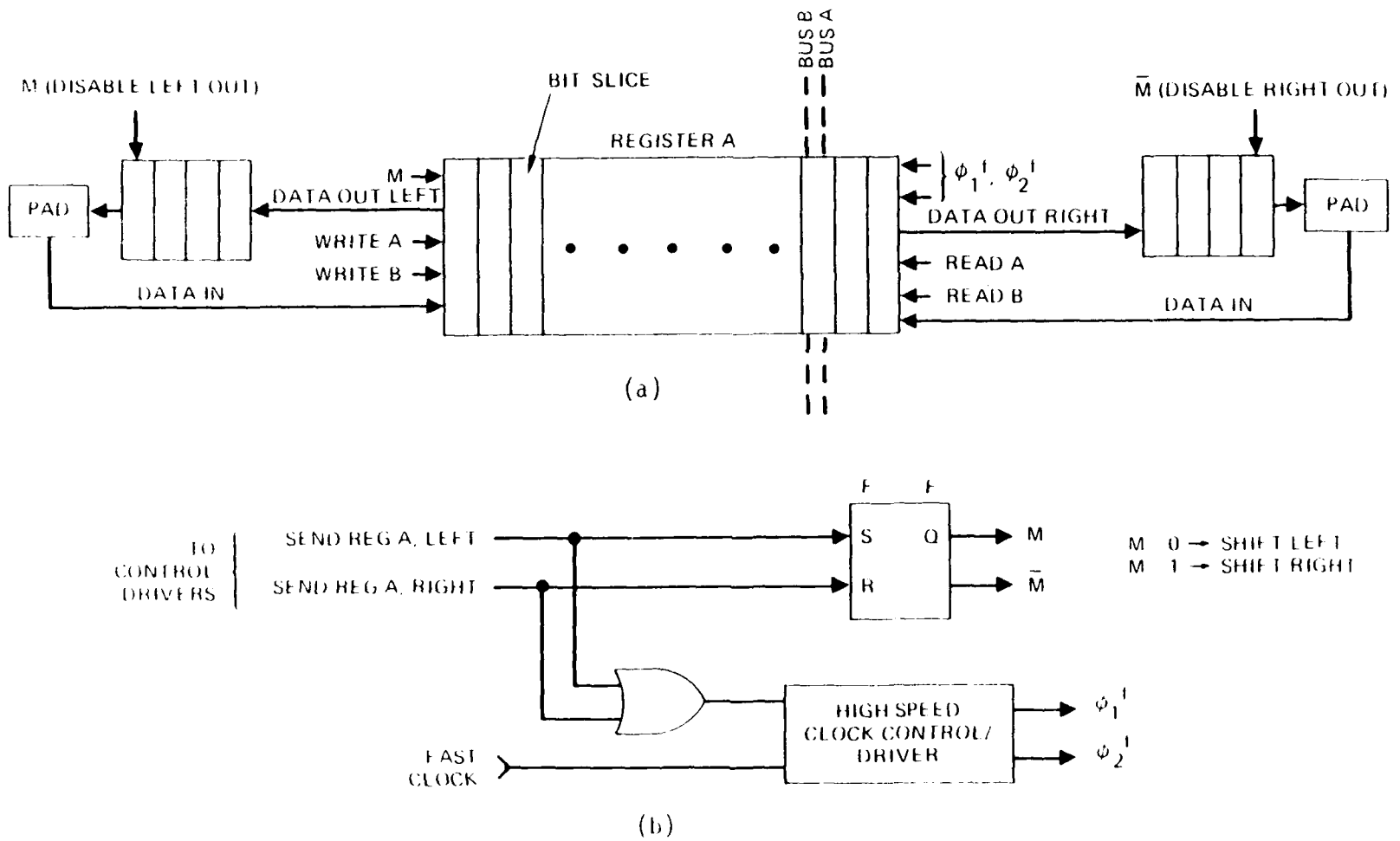


Figure 5. (a) Functional organization of a single I/O port and (b) some of the control circuitry.

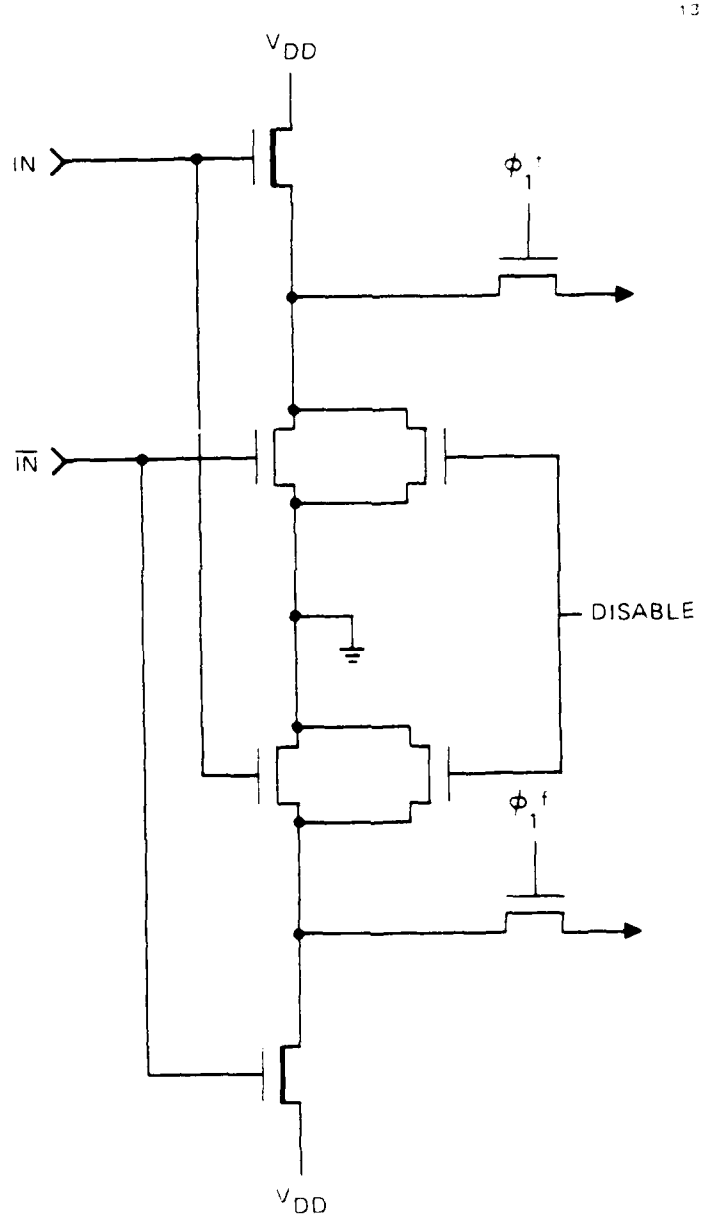


Figure 16. Pipelined driver stage for serial I/O port. The "disable" FETs are only included in last buffer stage. The fast clocks are ϕ_1, and ϕ_2.

appropriate I/O function module (i.e., the one which has a hardwired connection to the desired adjacent PE) and load the desired word into this module, if it wasn't there to begin with. A second set of control lines sends one of two signals (Reg A left or Reg A right in Figure 15(b)) to the function module which initiates transfer of data. Of course appropriate signals would have to be simultaneously sent to the adjacent chip so that it can receive the data word. The high speed clock driver/control circuitry then generates the appropriate number of clock cycles to transfer the data and then shuts itself off. At this point the data is now in the appropriate function module on the adjacent PE. In addition there is a new word in the register from which the data was originally sent. All I/O ports receive words while sending them.

A logic level diagram of several bit-slices of an I/O port is shown in Figure 17. Each bit-slice consists of a D-type FF and three gates used to direct the flow of data either to the left or right. The corresponding circuit level diagram for one of the bit-slices is shown in Figure 18. Six control lines plus two of the high speed clocks would be used to operate each bit-slice.

A possible problem regarding the transfer of information serially between PE's has to do with synchronization of the operation. Since this transfer can take place between two I/O units widely separated physically, skewing of the high speed clock could prevent correct operation. One solution to this is to use some handshaking scheme, although this would certainly degrade performance and increase hardware overhead. Alternatively, one could slow the operation down or transfer some of the bits in each word in parallel.

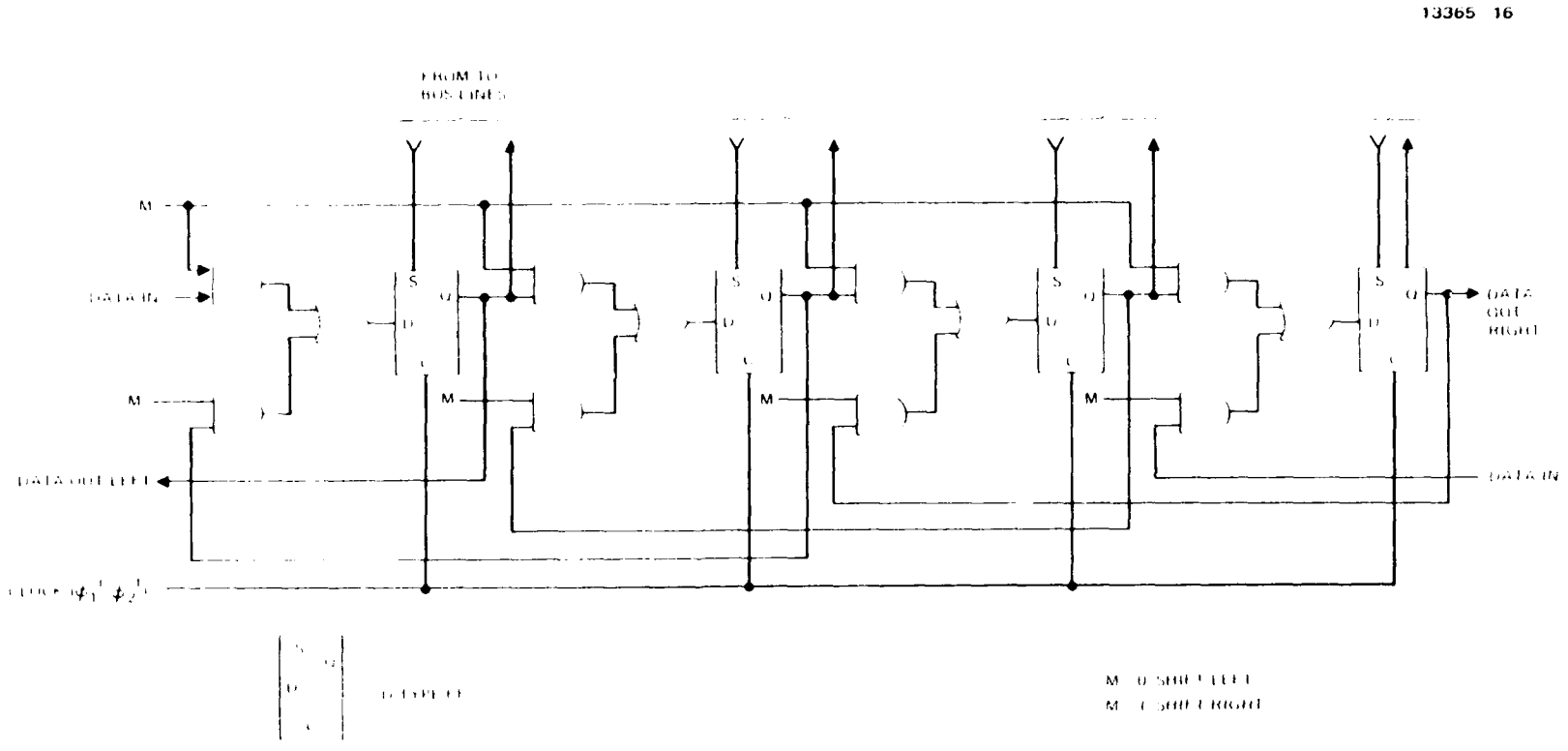


Figure 17. Logic diagram for shift-right, shift-left register and I/O port.

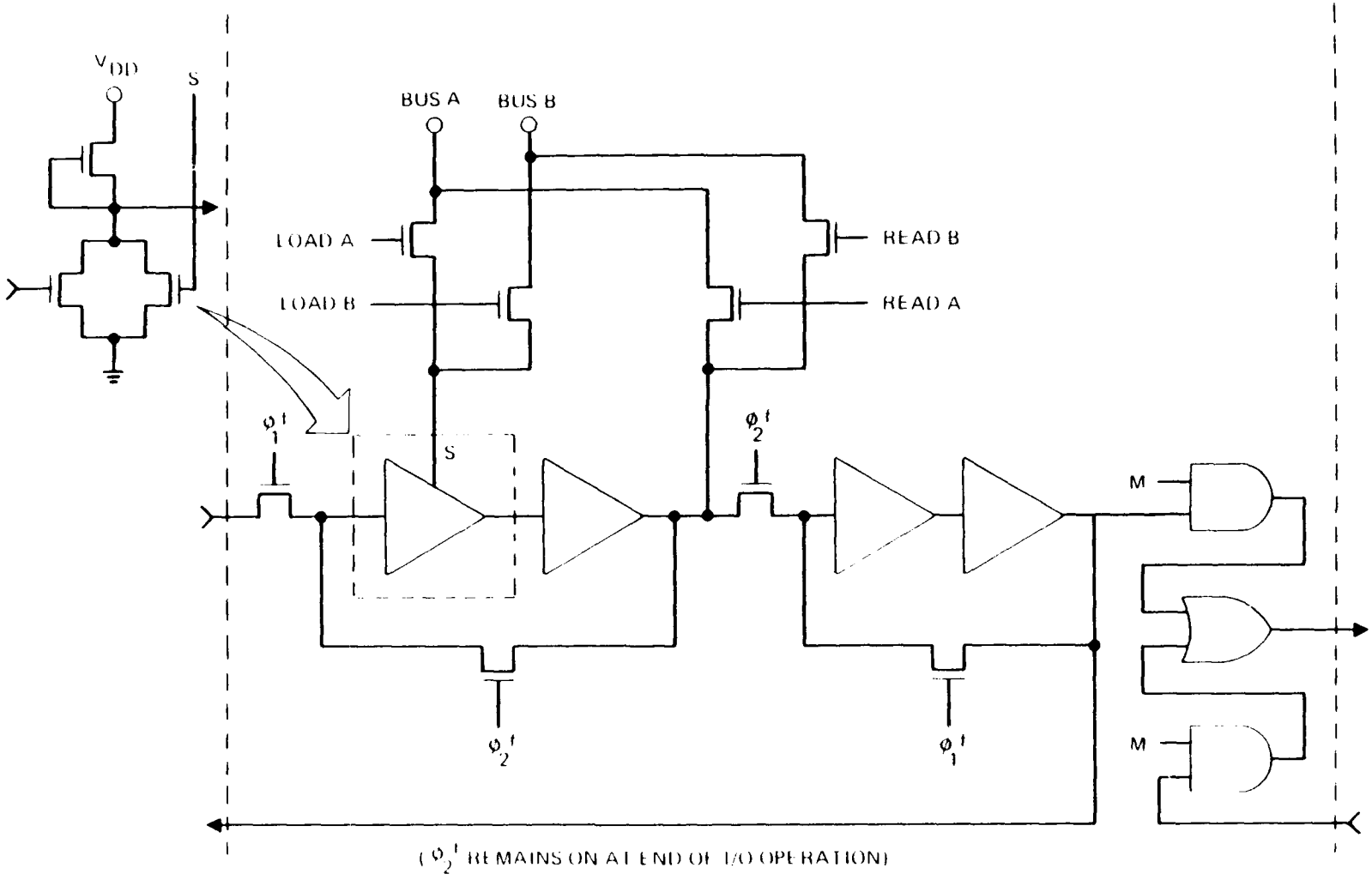


Figure 13. Register cell circuit diagram for serial I/O port. Here ϕ_2 and ϕ_1 refer to the two phase test clocks.

II-F. Division

There are numerous algorithms for performing division, and for implementing each of these there are many possible hardware schemes. In this section we will focus on some of the schemes which we feel are the most promising, while briefly describing some other alternatives.

Our basic goal was to identify an algorithm that would allow division to be performed on a linear array of carry-save type cells at a rate equal to that of multiplication. Since our multiplier is pipelined with a cycle time associated with only a couple of gate delays, this was a difficult task. Although we have identified a number of possible divider designs based on a carry-save type cell, it is not clear yet whether the desired speed can be obtained. This is basically due to two reasons: first, one does not know the quotient ahead of time, whereas for multiplication the multiplier is always known; this prevents us from pipelining the operation. Second, a large amount of time is generally needed in order to select a quotient digit. Although we can do little about the first problem, we have identified a couple of attractive schemes for simple quotient digit selection (truncation) that appear suitable for high speed divider implementation. We estimate that the best speeds obtainable will be between 1 and 3 times that of the multiplier.

The algorithms we have looked at are

Direct Methods

Multiplicative Normalization (MN)

Combined Multiplicative Normalization and Self Restoring Techniques

Iterations Based on Newston-Raphson Formula

CORDIC

we feel that the first three of the five are the most appealing from the point of view of the VLSI generic type implementation we are seeking. These are discussed more fully below.

Direct Methods

The normal pencil and paper approach to division is a trial and error method. The advantage of the direct algorithms is that the trial and error part of the algorithm has been replaced by a simple recursion that obtains the result in a given number of steps.[3] As in multiplication, much time can be wasted adding and subtracting partial remainders due to the limitations of carry propagation. It would be more desirable to use carry-save adders and subtractors. However, direct division in its simplest form requires information as to the sign of the remainder to select the quotient digit. The carry-save result would not provide this information. An alternative scheme is to use a redundant number representation[4,5], e.g., $q = -1,0,1$ for radix-2. With this approach one can still use a carry-save technique for evaluation of partial remainders. The quotient digit selection is based on the first 3 (radix-2) or 7 (radix-4) most significant bits of the partial remainders (a small carry propagate adder (CPA) is used for just these bits). For the radix-2 case the recursion is

---

3. Edward Braun, Digital Computer Design, Academic Press, N.Y., 1963.
4. J. E. Robertson, "A New Class of Digital Division Methods," IRE Trans. on Elect. Computers, EC-7, pp.218-222, Sept. 1958.
5. Milos Ercegovac, Private Communication.

$$R_{j+1}=2R_j-q_{j+1}x$$

$R_j=j^{th}$ partial remainder

x=divisor

$q_j=j^{th}$ quotient digit (redundant number representation)

R_0=dividend

and the selection rules for the quotient are

$$q_{j+1}=\begin{cases} 0 & -1/4\leq 2\hat{R}_j<1/4 \\ \text{sign} & \text{otherwise} \end{cases}$$

$$\text{sign}=\begin{cases} + & \text{if } \text{sign}(\hat{R}_j)=\text{sign}(x) \\ - & \text{if } \text{sign}(\hat{R}_j)\neq\text{sign}(x) \end{cases}$$

where $\hat{R}_j$ is the CPA output.

A simplified functional block diagram of an implementation is shown in Figure 19. The circuit is initialized by loading the dividend into the carry-save subtractor. The partial remainder estimate R_j supplied to the 3-bit CPA is used to generate the first quotient digit. This result is used to determine whether -1, 0, or 1 times x is to be subtracted during the next cycle. After n+1 cycles, q_n has been obtained. The final quotient result must be sent to a carry propagate adder in order to eliminate its redundant form.

The advantages of this division scheme are in its simplicity, regularity and fit to our bit-slice approach to processor design. The primary

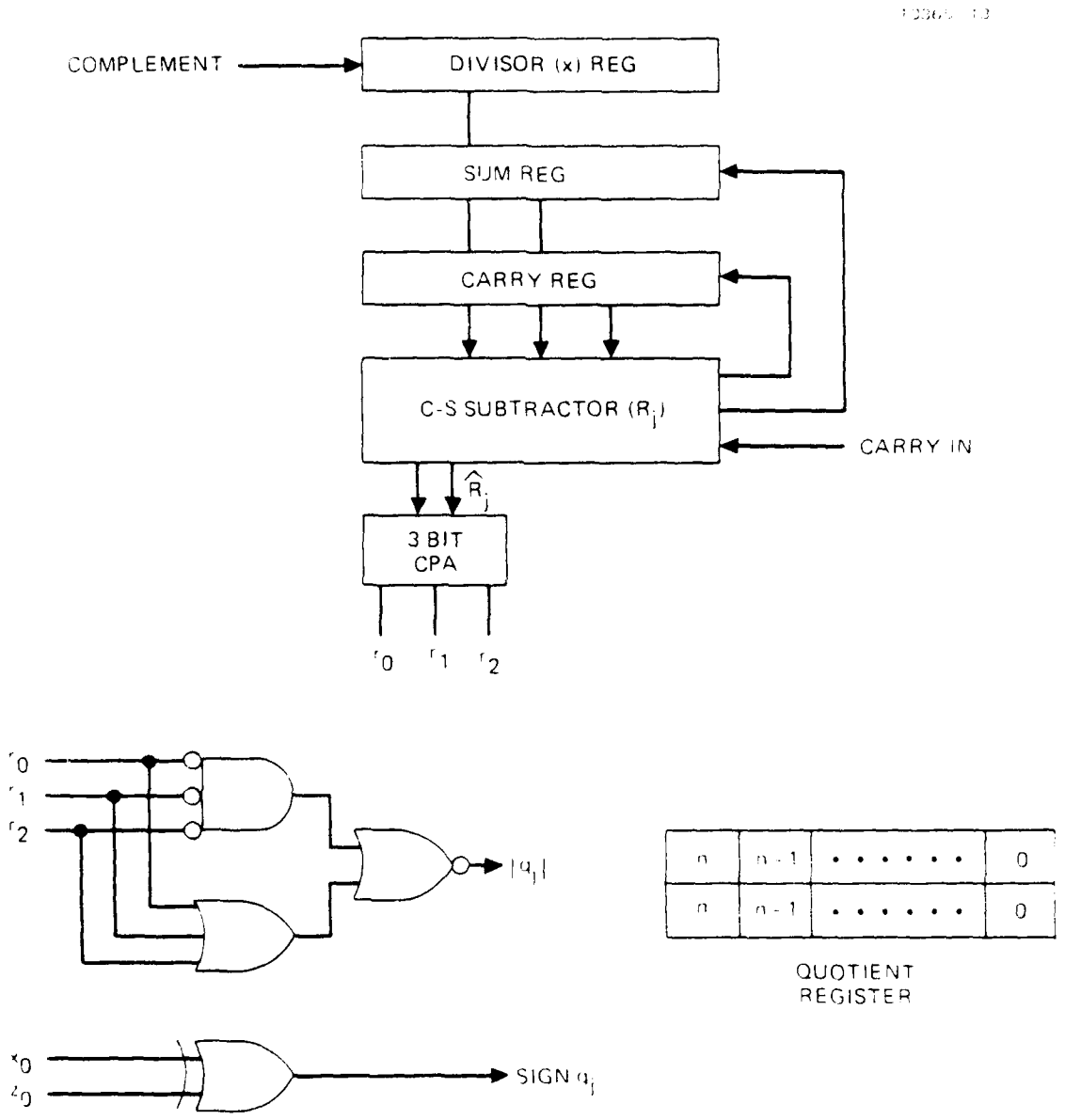


Figure 19. Function block diagram of a radix-2 implementation of the direct division algorithm. The quotient selection circuit is shown at bottom. Here $\hat{R}_j$ represents the first three bits of R_j.

disadvantage is its lack of speed. It can't be pipelined because the carry-save unit can't operate until the 3-bit CPA and the quotient circuit have finished, a total of approximately 5-6 gate delays. In addition this is only a radix-2 algorithm, whereas the multiplication is a radix-4 algorithm. A radix-4 implementation[4] would only require half the number of partial subtractions, but the quotient digit selection would be much more difficult.

Multiplicative Normalization

For the division y/x, if one can introduce a sequence of multiplications, M, such that Mx=1, then the same sequency applied to y will yield the desired quotient. This procedure, called multiplicative normalization (MN), has been mechanized in a way that requires multiplication by $(1+s_k 2^{-k-1})$, which can be done using only additions and shifts.[6] This approach uses the recursions

$$x_{k+1} = x_k(1 + s_k 2^{-k-1}), \quad x_k \longrightarrow 1, \quad 0 \leq k < n$$

$$y_{k+1} = y_k(1 + s_k 2^{-k-1}), \quad y_k \longrightarrow y/x \text{ as } k \longrightarrow$$

$$x_0 = x, y_0 = y, \quad 0.5 \leq x, y < 1.$$

In order to find s_k it is more convenient to work with scaled remainders,

$$R_k = (x_k - 1) \quad 2^k$$

$$R_{k+1} = 2R_k + s_k + s_k R_k 2^{-k}$$

where for $k \geq 1$,

---

6. B.G. deLugish, "A Class of Algorithms for Automatic Evaluation of Certain Elementary Functions in a Binary Computer," Dep. Comput. Sci., Univ. of Illinois, Urbana, IL, Rep. 399, June 1, 1970.

$$s_k = \begin{cases} 1, & \text{if } R_k < -3/8 \\ -1, & \text{if } R_k \geq 3/8, \\ 0, & \text{otherwise.} \end{cases}$$

and for k=0

$$s_0 = \begin{cases} 2, & \text{if } -1/2 \leq R_0 < -1/4 \\ 0, & \text{if } -1/4 \leq R_0 < 0. \end{cases}$$

A functional implementation for these equations is illustrated in Figure 20. Here, there are two separate sets of hardware, one for recursion in R_k and the other for the quotient y_k. Only an estimate of R_k is required at each recursion so that a small 3-bit CPA is required along with a few gates to determine s_k. This approach also requires a variable shifter network for both of the two hardware sections.

The MN approach is very similar to the direct methods in that they both use carry-save circuits and the selection operation for q_j or s_k is very similar. If fast shifter networks can be built, then the speeds of the two algorithms will be approximately the same. The major difference between the two is that the MN approach uses approximately three times the hardware. However, there is an advantage to MN in that there is far more generality in its capabilities, which include multiplication, square root, logarithm, exponentials, trigonometric functions and inverse trigonometric functions.[7]

---

7. Milos D. Ercegovac, "Radix-16 Evaluation of Certain Elementary Functions," IEEE Trans. on Computers, C-22, pp.561-566, June 1973.

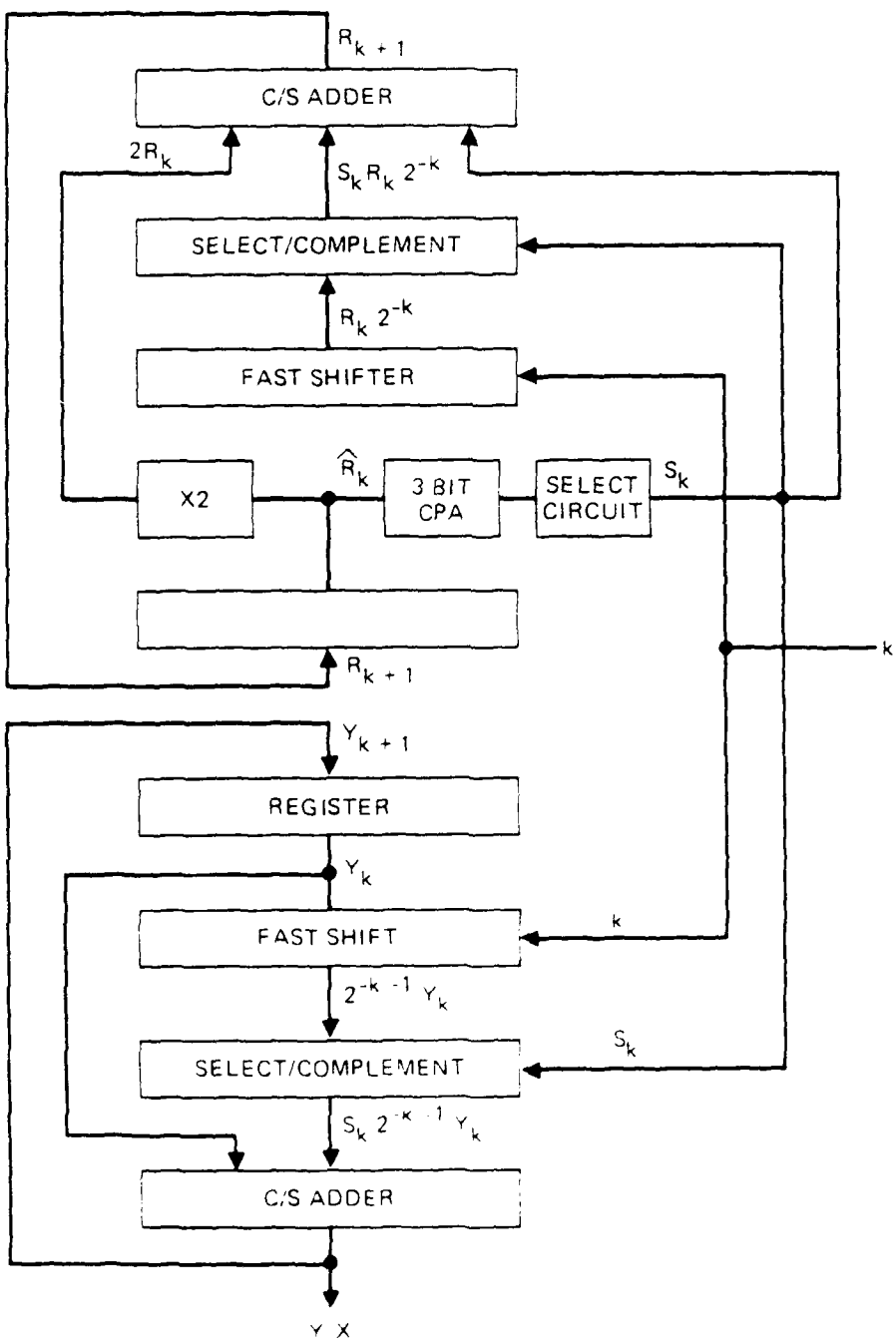


Figure 20. Function block diagram for multiplicative normalization division algorithm. Here $\hat{R}_k$ refers to the first three digits of R_k.

Combined Multiplicative Normalization and Non-restoring Division

Although the previously described algorithms are reasonably fast to fit well into our bit-slice, carry-save, serial-parallel MOP chip organization, they still do not satisfy our basic goal of having a divider that operates at the same speed as our MOP chip multiplier. We feel that the best approach to achieving this goal would be to use a higher radix algorithm for division. This effectively allows us to reduce the number of recursions by k, where $r=2^k$ is the radix. The problem with this approach is that while the number of recursions is reduced, the selection procedure for q_j or s_k becomes increasing complex, increasing the time required at each recursion. For example a radix-4 implementation of the direct method[4] requires as input to the quotient selection circuit the first seven bits of the partial remainder. Thus, the CPA addition takes longer and the quotient selection circuit is more complex as well.

A promising alternative to this problem has been suggested[8,9] which combines both MN and direct methods (see Appendix A). In this scheme MN is used to transform both the dividend and divisor into a range which allows the quotient digits to be selected by simple truncation of the partial remainders. Limited CPAs can be used to form the most significant part of the partial remainder with the quotient select circuit replaced by a simple truncation circuit. For a radix-4 implementation of this circuit, the speed could be increased by a factor of at least two. The basic recursions for this algorithm (radix-4) are

---

8. Milos D. Ercegovac, "A Higher-Radix Division with Simple Selection of Quotient Digits," 6th IEEE Symposium on Computer Arithmetic, Denmark, 1983.
9. Milos D. Ercegovac, "Division Schemes with Simplified Selection Rules and Prediction of Quotient Digits," Unpublished Report, August 3, 1983.

$$R_{j+1}=4(R_j-q_j x^*)$$

$$q_{j+1}=\text{Trunc}[4(R_j-q_j+c)]$$

where

$$c = \begin{cases} 1/2 & \text{if } R_j \geq q_j \\ -1/2 & \text{otherwise} \end{cases}$$

and

x^* = transformed divisor

The minimum time step required to execute this algorithm is the time to compute $q_j x^*$ plus the time to compute R_{j+1}. Unfortunately, it is not possible to pipeline these calculations so that this radix-4 algorithm can not be executed as fast as that for multiplication. However, a radix-8 implementation looks promising.

CORDIC Algorithm

The CORDIC algorithm[10,11] is well known for the wide variety of elementary functions which it can evaluate. Modifications have been suggested to speed up the algorithm and incorporate floating point operands.[12] To implement this algorithm requires three adder/subtractor units, a ROM to store n integers (n=bit length), plus a couple of shifters. At each of n recursions, three "ax+b" type calculations are performed. Finally, a scaling operation is sometimes necessary.

The drawback of this algorithm is the considerable amount of hardware

---

10. J.E. Volder, "The Cordic Trigonometric Computing Technique," IRE Trans. on Electronic Computers, EC-8, pp.330-334, Sept. 1959.
11. J.S. Walther, "A Unified Algorithm for Elementary Functions," 1971 Spring JCC, pp.379-385.
12. H.M. Ahmed, PhD Thesis, Stanford, 1980.

required to implement it. (It requires more than three times the hardware used in the direct method.) In its basic form it is also slow; however, use of a higher radix and introduction of redundant numbers could provide the desired speeds.

Iterations Based on Newton-Raphson Method

The Newton-Raphson equation allows one to find progressively more accurate solutions to the equation $f(x)=0$ using the formula

$$x_{i+1}=x_i-[f(x_i)/f'(x_i)] \qquad (1)$$

For the case of division $f(x)=(1/x)-s$, where s is the reciprocal of x. This gives $x_{i+1}=x_i(2-sx_i)$.

One popular variation on this method is the Goldschmidt algorithm[13], which was implemented on the IBM 360 Model 91. If $y/d=q$ and we find some number k such that $kd=1$, then $ky=q$. The number k represents a sequence of multiplications by $(2-x_k)$, where $x_{k+1}=x_k(2-x_k)$ and $x_0=d$, corresponding to the case where $s=1$ in the Newton-Raphson formula above. If d is normalized ($d\geq 1/2$) and we let $d=1-x$, then $x_0=1-x$, $x_1=1-x^2$, and $x_n=1-x^{2n}$, so that x_n converges to unity quadratically. Then, if $y_{k+1}=y_k(2-x_k)$, $y_0=y$, then y_{k+1} will approach q quadratically as d approaches 1. The quadratic convergence is particularly useful for large words, because each iteration doubles the number of known bits. No remainder is generated, however.

The principal problem with the iteration techniques is that they require several passes through a multiplier, making them necessarily slower than

---

13. R. E. Goldschmidt, "Application of Division by Convergence," M.S. Thesis, MIT, Cambridge, MA, June 1964.

desired. (Pipelining can be used for this algorithm to speed it up, however, with the addition of much additional hardware.) Moreover, it is not suitable for integration into our bus oriented, bit-slice chip organization, because the multipliers must be fast parallel implementations with the special busing hardware to permit rapid data movement between iterations. (Parallel multipliers are not well suited to bit-slice organizations.)

II-G. Square Root

Numerous algorithms exist for evaluating square roots[3,5,7,11,14,15], manyh commonality between the problem of division and square root, we will only briefly discuss this topic.

Techniques Based on Newton-Raphson Iterations

Iterative techniques are perhaps best known for solving the square root problem and are reasonably fast due to their quadratic convergence. If one uses $f(x)=x^2-a$ in Equation (1), then the iterative equation for the square root is

$$x_{n+1}=[x_n+(a/x_n)]/2$$

The disadvantage of this formulation is that a division operation is required every iteration. An alternative formulation is to find the reciprocal of the square root. This uses the iteration

---

14. T. Chi Chin, "Automatic Computation of Exponentials, Logarithms, Ratios and Square Roots," IBM J. Res. Develop., July 1972, 380-388.
15. M. D. Ercegovac, "An On-Line Square Rooting Algorithm," Proc. Fourth IEEE Symposium on Computer Arithmetic, Oct. 1978, pp. 183-189.

$$x_{n+1}=3x_n-(x_n)^3/(2a)$$

More multiplies are required each iteration; however, division is required only once.

Odd Series Approximation

This technique is based on the observation that the square root of the sum of a series of odd numbers: 1, 3, 5, . . . has a value that corresponds to the position of the highest term in the series[3]. For example, the sum of 1, 3, 5, and 7 is 16 and the square root of 16, which is 4, corresponds to the position of 7 in the series 1, 3, 5, 7.

The square root extraction procedure can be reduced to three steps:

(1) Separate the bits of the radicand into groups of two bits each, starting from the binary point.

(2) Begin the actual extraction operation at the first group of bits from the left that does not contain two zeroes. Align a "1" with the right-hand bit of this group and subtract. The remainder will be nonnegative and a "1" is entered in the root for this group. For each double "0" group to the left of this group, a "0" is entered in the root.

(3) For all succeeding groups, the trial factor to be subtracted from the remainder is the expression $(4r_{n-1}+1)$, where r_{n-1} pertains to the result obtained up to the (n-1)th iteration. The right hand digit of the trial divisor is aligned with the right hand digit of the group for which it is used, and subtracted. If the remainder is nonnegative, a "1" is entered as the root for that group. If the remainder is negative, the root is "0" and the subtraction is restored.

An example of square root extraction is given in the example below.

```
 . 1      1      0      1
√.10     10     10     01
  1
 --
  1      10
  1      01  = (4 x 1) + 1
 ---------
         1      10
        11      01  = (4 x 3) + 1
        --------
        Restore
         1      10      01
         1      10      01  = (4 x 6) + 1
        ---------------------
```

Check: .10101001 = 169/256
.1101 = 13/16

This algorithm has some important advantages over the direct division approaches described in the previous section. Most important is that the selection of the result for each group is very simple, just a test for a negative result. Keeping in mind that it is not necessary to store a negative result, the need for restoration will not cost much in terms of speed (this is equivalent to non-performing division).

APPENDIX A

I. A HIGHER-RADIX DIVISION WITH SIMPLE SELECTION OF QUOTIENT DIGITS

Milos D. Ercegovac
UCLA Computer Science Department
University of California, Los Angeles

3732-C Boelter Hall
(213)825-2660

ABSTRACT

A higher-radix division algorithm with simple selection of quotient digits is described. The proposed scheme is a combination of the multiplicative normalization used in the continued-product algorithms and the recursive division algorithm. The scheme consists of two parts: in the first part, the divisor and the dividend are transformed into the range which allows the quotient digits to be selected by rounding partial remainders to the most significant radix-r digit in the second part. Since the selection requires only the most significant part of the partial remainder, limited carry-propagation adders can be used to form the partial remainders. The divisor and dividend transformations are performed in three steps using multipliers of the form $1 + s_k r^{-k}$ as in continued product algorithms. The higher radix of the form $r = 2^k$, $k=2,4,8,...,$ can be used to reduce the

number of steps while retaining the simple quotient selection rules.

I. INTRODUCTION

In this article a division scheme characterized by a simple method for selecting quotient digits is described. The scheme also has several properties important for modular implementation. Division algorithms have been of a wide interest [ROBE58, METZ62, ATKI68, ANDE68, TAYL81] because of the problems of fast and efficient selection of quotient digits and computation of partial remainders, and compatibility of implementation with other more frequent arithmetic operations such as multiplication.

The scheme for division suggested here consists of two parts. In the first part the divisor X is forced into a suitable range and the dividend Y is adjusted. The divisor and dividend transformations are performed using a few initial steps of the iterative multiplicative normalization algorithm [ERCE73, DELU70]. In the second part the quotient digits are obtained by a recursive algorithm [ERCE75, ERCE77] in which the selection can be performed by rounding. The proposed division scheme generates an m-digit quotient in m+3 additive steps which do not require full precision carry propagation. The scheme also provides the remainder.

The division schemes based on the range transformation have been considered before [SVOB63, KRIS70, ERCE75]. The main contributions of this article are implementation-efficient transformation and a simple quotient selection method.

In Section II a derivation of the division scheme is presented. A radix-16 division algorithm is given in Section III. The implementation aspects are discussed in [ERCE83].

II. DERIVATION OF THE DIVISION SCHEME

Consider the division problem

$$Y = XQ + R \tag{1}$$

where

X is the n-bit divisor, $|X| \in [1/2, 1)$;

Y is the 2n-bit dividend, $|Y| < |X|$;

Q is the n-bit quotient and

R is the corresponding remainder.

A binary recursive division algorithm computes sequentially the partial remainders and the quotient digits using the recursion

$$R_{j+1} = 2R_j - q_{j+1}X, \quad j=0,1,2,\ldots,n-1 \tag{2}$$

where

$R_0 = Y$ is the initial remainder,

$q_{j+1} = f(R_j, X)$ is the j+1-th quotient bit, and

f is a selection function.

In order to reduce the number of steps, the binary algorithm can be modified so that b bits of the quotient are obtained per step. That is, the radix of implementation is defined to be $r = 2^b$. However, the use of a higher radix makes the selection of the quotient digits as well as the computation of the partial remainders more complex [ROBE58, ATKI68].

We now describe a division algorithm in which the selection can be performed by a simple rounding.

Recursion and Selection

The recursive algorithm for division in which the quotient digits are obtained by rounding partial remainders to the integer part and taking the integer part as the quotient digit requires the divisor to be in the range

$$[\ 1 - \alpha,\ 1 + \alpha\ ] \tag{3}$$

where α is a constant between 0 and 1, to be determined later. It also requires the use of a redundant representation of the quotient digits. A symmetric redundant digit set (signed-digit set [AVIZ61]) is used:

$$D_\rho = \{-\rho, \ldots, -1, 0, 1, \ldots, \rho\} \tag{4}$$

where

$$r/2 \leq \rho < r \quad \text{and } r \text{ is the radix.}$$

The recursion is

$$R_j = r(R_{j-1} - q_{j-1}X') \tag{5}$$

and

$$q_j = \text{SELECT}(R_j) \tag{6}$$

$$= \begin{cases} \text{sign } R_j \left\lfloor |R_j| + \frac{1}{2} \right\rfloor & \text{if } |R_j| \le \rho \\ \text{sign } R_j \left\lfloor |R_j| \right\rfloor & \text{otherwise} \end{cases}$$

where

R_j is the j-th remainder;

X' is the scaled divisor such that

$$1 - \alpha \le |X'| \le 1 + \alpha,$$

and

$q_j \leftarrow D_\rho$ is the j-th quotient digit.

Initially,

$$R_0 = Y'$$

is the scaled dividend Y such that

$$|R_0| \le \rho + \beta \text{ and } 1/2 \le \beta < \frac{\rho}{r - 1} \tag{7}$$

The validity of the recursion and the selection function is established by proving the following two claims.

Claim 1:

If the bound α is

$$0 < \alpha \leq \frac{1}{r}\left[1 - \frac{\beta(r-1)}{\rho}\right] \tag{8}$$

and q_j is the j-th quotient digit from a signed-digit set $D_\rho = \{-\rho,\ldots,-1,0,1,\ldots,\rho\}$, $r/2 \leq \rho < r$, selected according to the function SELECT, then the partial remainder R_j satisfies

$$|R_j| \leq \rho + \beta \tag{9}$$

for all j.

Proof:

To show that the partial remainders are bounded we proceed by induction. By definition (7):

$$|R_0| \leq \rho + \beta$$

Assume

$$|R_{j-1}| \leq \rho + \beta$$

Let $A = 1 - X'$ so that $|A| = \alpha$. Then

$$|R_j| \leq r|R_{j-1} - q_{j-1}| + r|A||q_{j-1}| \tag{10}$$

$$\leq r(\rho + \beta - \rho) + r\alpha\rho$$

$$= r\beta + r\left[\frac{1}{r}\left(1 - \frac{\beta(r-1)}{\rho}\right)\right]\rho$$

$$= \rho + \beta$$

because, by definition of the selection function SELECT, the

choice of digit q_j can always be made such that

$$|R_j - q_j| \leq \beta \tag{11}$$

□

Claim 2:

Let $Q^* = \sum_{i=0}^{m} q_i r^{-i}$ be the computed quotient. Then

$$\left|\frac{Y'}{X'} - Q^*\right| \leq r^{-m} \tag{12}$$

Also, $R = r^{-m-1} R_{m+1}$.

Proof:

By substitution

$$Y' = X' \sum_{i=0}^{m} q_i r^{-i} + r^{-m-1} R_{m+1} \tag{13}$$

and

$$\left|\frac{Y'}{X'} - Q^*\right| \leq r^{-m-1} \frac{|R_{m+1}|_{max}}{|X'|_{min}} \tag{14}$$

$$= r^{-m-1} \left[\frac{\rho + \beta}{1 - \alpha}\right]$$

$$= r^{-m} \quad \text{for } \rho = r-1$$

$$< r^{-m} \quad \text{for } \rho < r-1$$

From (13, 14), $R = r^{-m-1} R_{m+1}$.

□

According to the analysis of the rounding selection method [ERCE75] the bounds α, β, ρ and the selection interval overlap Δ are related as follows. First, in order to have efficient implementation of single-digit multipliers, required by the division recursion, the maximum digit value should be [ATKI70]:

$$\rho \leq \frac{2(r - 1)}{3} \tag{15}$$

Therefore, from (7):

$$1/2 < \beta < 2/3$$

On the other hand, $\beta = \frac{1}{2}(1 + \Delta)$, where Δ is the overlap between the selection intervals [ERCE75]. Therefore, the upper bound on α can be written as:

$$\alpha \leq \frac{1}{r}[1 - \frac{3\beta}{2}]$$

$$= \frac{1}{r}[1 - \frac{3(1 + \Delta)}{4}]$$

For $\Delta = 1/r$,

$$\alpha \leq \frac{r - 3}{4r^2} \tag{16}$$

This bound will be used to define the range of the transformed divisor.

To transform the divisor into this range and adjust the dividend Y, we adopt the multiplicative normalization technique [DELU70, ERCE73].

Range Transformations

The multiplicative normalization of a given argument

$$|X_0| \in [1/2,1)$$

is defined as a sequence of transformations such that

$$1-\alpha \le |X_0 \prod_{i=0}^{p-1} M_i| \le 1+\alpha \tag{17}$$

for a given constant $0 \le \alpha < 1$ and the number of steps p. The multipliers are of the form $M_k = 1 + S_k r^{-k}$, where r is the radix and S_k is a digit in a redundant radix r number system. The form of the multipliers simplifies the implementation since the full-precision multiplication is replaced by an addition, a single radix-r digit multiplication and a k-position shift.

The multiplicative normalization is performed recursively:

$$X_{k+1} = X_k(1 + S_k r^{-k}), \quad 0 \le k < p \tag{18}$$

The digit value of S_k is chosen such that the error e_{k+1} after step k is

$$|e_{k+1}| = |1 - X_k(1 + S_k r^{-k})| \le \frac{\rho}{r-1} r^{-k} \tag{19}$$

The number of the transformation steps p can now be obtained from the following condition, implied by (16) and (19):

$$|e_p| \le \alpha \tag{20}$$

Assuming an overlap $\Delta = \frac{1}{r}$, it follows that, for $r \ge 8$, $p \ge 3$. That is, three steps are sufficient to transform given divisor X and dividend Y into the required range.

The multiplicative normalization is conveniently performed using a recursion on scaled differences (remainders). Let

$$D_k = r^{k-1}(X_k - 1), \quad 0<k<p \tag{22}$$

From (18) and (22), the scaled difference recursion follows:

$$D_{k+1} = rD_k + S_k + S_k D_k r^{-k+1}, \quad 0<k<p \tag{23}$$

For p=3, the normalization procedure requires determination of S_0, S_1 and S_2. A complete derivation procedure for the selection rules is discussed in [ERCE72]. For the sake of brevity, we only show the radix-16 rules in the next section.

III. RADIX-16 ALGORITHM

In this section the division scheme is illustrated for r=16. The algorithm is as follows:

```
/* Part 1 - Range Transformation
/* Inputs:  Divisor X0 ← [1/2,1)
/*          Dividend Y0, |Y0|<|X0|
/* Outputs: Transformed divisor X'
/*          Transformed dividend Y'

   1:  if 1/2≤X0<5/8
       then
         D1 ← 2X0 - 1
         Y1 ← 2Y0
       else
```

$D_1 \leftarrow X_0 - 1$

$Y_1 \leftarrow Y_0$

2: $S_1 \leftarrow \overline{sign}D_1\lfloor 16(D_1 + U_1)\rfloor$

3: $D_2 \leftarrow 16D_1 + S_1 + S_1D_1$

$Y_2 \leftarrow Y_1(1 + S_1 16^{-1})$

4: $S_2 \leftarrow \overline{sign}D_2\lfloor 16(D_2 + U_2)\rfloor$

5: $X' \leftarrow (16D_2 + S_2 + S_2D_2 16^{-1} + 1)16^{-2}$

$Y' \leftarrow Y_2(1 + S_2 16^{-2})$

$q_{-1} = 0$

/* Part 2 - Division Recursion

/* Inputs: Divisor X'

/* Dividend Y'

/* Outputs: Quotient $Q^* = \sum_{i=0}^{m} q_i 16^{-i}$

/* Remainder $R = 16^{-m-1}R_{m+1}$

7: For j = 0,1,2,..., m do

7.1: $R_j \leftarrow 16(R_{j-1} - X'q_{j-1})$

7.2: $q_j \leftarrow SELECT(\hat{R}_j)$

END

The selection function SELECT, defined in (6), is performed on an estimate $\hat{R}_j$ of the partial remainder such that $|\hat{R}_j - R_j| \leq \frac{1}{16}$. The terms U_1 and U_2 are six-bit rounding constants defined as functions of the seven leading bits of the truncated scaled difference D_j, j=0,1,2.

$$U_k = \sum_{i=1}^{6} u_i 2^{-i} \qquad (24)$$

where the switching expressions for u_i's are

$$u_1 = u_2 = 0,$$

$$u_3 = K1 d_0 \bar{d}_2,$$

$$u_4 = K1 d_0 \bar{d}_4 (\bar{d}_2 + \bar{d}_3),$$

$$u_5 = K1(d_0 + \bar{d}_3 \bar{d}_4) + K2[d_0 + \bar{d}_1(\bar{d}_2 + \bar{d}_3) + d_6]$$

$$u_6 = K1 \bar{d}_3 d_4 + K2 d_0 (d_1 + d_2 d_3)$$

and K1 and K2 denote steps 1 and 2, respectively. The derivation of these step-dependent rounding constants is based on the selection intervals given in the Appendix. More detailed discussion can be found in [ERCE72].

An example of division is given in Figure 1.

IV. CONCLUSION

A scheme for division has been presented. It consists of a 3-step transformation of the divisor and the dividend into a range which allows use of a recursive higher-radix division algorithm with a simple quotient selection method. A detailed derivation of the range transformation requirements and the procedure has been described and an algorithm for r=16 has been given. The implementation details and the performance are discussed elsewhere [ERCE83].

REFERENCES

[ANDE68] S.F. Anderson, J.G. Earle, R.E. Goldschmidt, and D.M. Powers, "The IBM/360 Model 91: Floating-Point Execution Unit", IBM Journal, January 1967, pp.34-53.

[AVIZ61] A. Avizienis, "Signed Digit Number Representations for Fast Parallel Arithmetic", IRE Trans. on Electronic Computers, 1961, pp.389-400.

[ATKI68] D. E. Atkins, "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders", IEEE Trans. on Computers, Vol.C-17, No. 10, October 1968, pp.925-934.

[ATKI70] D. E. Atkins, "Design of the Arithmetic Units of ILLIAC III: Use of Redundancy and Higher Radix Methods", IEEE Trans. on Computers, August 1970, pp.720-733.

[DELU70] B.G. DeLugish, " Class of Algorithms for Automatic Evaluation of Certain Elementary Functions in a Binary Computer", Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, June 1970.

[ERCE72] M.D. Ercegovac, "Radix 16 Division, Multiplication, Logarithmic and Exponential Algorithms Based on Continued Product Representations", MS Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, August 1972.

[ERCE73] M. D. Ercegovac, "Radix-16 Evaluation of Certain Elementary Functions", IEEE Trans. on Computers, June 1973, pp.561-566.

[ERCE75] M.D. Ercegovac, "A General Method for Evaluation of Functions and Computations in a Digital Computer", Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, August 1975.

[ERCE77] M. D. Ercegovac, "A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer", IEEE Trans. on Computers Vol. C-26, No. 7, July 1977, 667-680.

[ERCE83] M.D. Ercegovac, "Higher-Radix Division: Implementation Alternatives and Their Performance", (in preparation), 1983.

[KRIS70] E.V. Krishnamurthy, "On Range-Transformation Techniques for Division", IEEE Transactions on Computers, Vol.C-19, No.3, March 1970, pp.227-231.

[ROBE58] J. E. Robertson, "A New Class of Digital Division Methods", IRE Trans. on Electronic Computers, September 1958 pp. 88-92.

[SVOB63] A. Svoboda, "An Algorithm for Division", Information Processing Machines, Vol.9, 1963, pp.25-32.

[TAYL81a] G. S. Taylor, "Compatible Hardware for Division and Square Root", Proc. 5th Symposium on Computer Arithmetic, 1981, pp.127-134.

[TUNG68] C. Tung, "A Division Algorithm for Signed-Digit Arithmetic", IEEE Transactions on Computers, Vol.17, 1968, pp.887-889.

[ZURA81] J. H. P. Zurawski and J. B. Gosling, "Design of High-Speed Digital Divider Units", IEEE Trans. on Computers, Vol. C-30, No.9, September 1981, pp.691-699.

Appendix

The selection intervals for S_1 and S_2 are shown in Tables 1 and 2, respectively. The detailed procedure for the derivation is given in [ERCE72].

S_1	$64\underline{D}_1$	$64\bar{D}_1$
10	-26	-23
9	-24	-22
8	-23	-20
7	-21	-18
6	-19	-16
5	-17	-14
4	-14	-11
3	-12	-8
2	-9	-5
1	-6	-2
0	-2	3
-1	2	7
-2	7	12
-3	12	18

Table 1: Selection Intervals for S_1

S_2	$64\underline{D}_2$	$64\bar{D}_2$
10	-42	-36
9	-37	-33
8	-33	-29
7	-29	-25
6	-25	-21
5	-22	-18
4	-18	-14
3	-14	-10
2	-10	-6
1	-6	-2
0	-2	3
-1	2	6
-2	6	10
-3	10	14
-4	14	18
-5	18	23

-6	23	27
-7	27	31
-8	31	35
-9	35	39
-10	39	42

Table 2: Selection Intervals for S_2

```
Divisor  x0 = 0.8107509300,
Dividend y0 = 0.5990471500,
Quotient  Q = 0.7388793868

Part 1:

   After Step 1: d1= -0.1892490700, y1= 0.5990471500, s1 = 4
   After Step 2: d2=  0.2150186000, y2 =0.7488089375, s2 = -3

Transformed divisor and dividend:

   X'= 1.0015624282, Y'= 0.7400338328

Part 2:

      i         Remainder   q          Quotient            Error

      1     -4.1844575266    1     1.0000000000    -0.2611206132
      2     -2.8513250219   -4     0.7500000000    -0.0111206132
      3      2.4537962023   -3     0.7382812500     0.0005981368
      4      7.2107415359    2     0.7387695313     0.0001098555
      5      3.1968726195    7     0.7388763428     0.0000030440
      6      3.0749653602    3     0.7388792038     0.0000001830
      7      1.1244492114    3     0.7388793826     0.0000000042
      8      1.9661885316    1     0.7388793863     0.0000000005

(All numbers are represented in decimal)
```

Figure 1: Example

II. DIVISION SCHEMES WITH SIMPLIFIED SELECTION RULES AND PREDICTION OF QUOTIENT DIGITS

Milos D. Ercegovac

August 3, 1983

Report No.1

1. Introduction

In a previous report, a paper presented at the 6th IEEE Symposium on Computer Arithmetic [ERCE83], a general division scheme was presented, based on a divisor/dividend transformation technique such that the selection of the quotient digits can be performed by simple rounding.

In this report we elaborate on the implementation and performance aspects of a radix-4 variant. Of particular interest is the fact that the next quotient digit can be obtained in parallel with the next remainder computation.

The discussion and results discussed here are preliminary and require further refinement.

2. Divisor and Dividend Transformation

We follow closely the results from [ERCE83] in this derivation.

Bound on the divisor

$$\alpha \leq \frac{r-3}{4r^2} = \frac{1}{64} > 0.0156$$

so that the transformed divisor X^* is in the interval $[1-1/64, 1+1/64]$

Transformation steps

The scaled remainders for the transformation are defined as

$$D_k = 4^{k-1}(X_k - 1)$$

where $X_0 = X$. We want that $|X_p - 1| \leq 1/64$ or, equivalently, that $D_p 4^{-p+1} \leq 1/64$. Assuming that $|D_p| \leq 1$, $p=4$.

The expressions for the transformation are:

$$D_1 = X_1 - 1 = \begin{cases} 2X_0 - 1 & \text{if } X_0 < 0.75 \\ X_0 - 1 & \text{otherwise} \end{cases}$$

That is, $S_0 \leftarrow \{0,1\}$.

$$D_2 = 4D_1 + S_1 + S_1 D_1$$

Equivalently,

$$D_2 = \begin{cases} 5D_1 + 1 & \text{if } S_1 = 1 \\ 4D_1 & \text{if } S_1 = 0 \\ 3D_2 - 1 & \text{if } S_1 = -1 \end{cases}$$

$$D_3 = 4D_2 + S_2 + S_2 D_2/4$$

$$D_4 = 4D_3 + S_3 + S_3D_3/16$$

The transformed divisor is

$$X^* = X_4 = D_4 4^{-3} + 1$$

The initial dividend is transformed using the following recursion:

$$Y_{k+1} = Y_k(1 + S_k 4^{-k}) \qquad k=0,1,2,3$$

Selection of S_1, S_2 and S_3

The selection intervals are determined by evaluating

$$D_k = (D_{k+1} - S_k(4 + S_k 4^{-k+1})$$

for D_{k+1} = dmax/dmin and all values of S_k = -2,-1,0,1,2. Assuming $-0.99 < D_4 < 0.99$ we obtain the following intervals:

dmin = -0.99, dmax = 0.99

Selection Intervals for k= 3

s =	-2,	dmin =	0.2606452,	dmax =	0.7716129,	delta =	0.7716129
s =	-1,	dmin =	0.0025397,	dmax =	0.5053968,	delta =	0.2447517
s =	0,	dmin =	-0.2475000,	dmax =	0.2475000,	delta =	0.2449603
s =	1,	dmin =	-0.4898462,	dmax =	-0.0024615,	delta =	0.2450385
s =	2,	dmin =	-0.7248485,	dmax =	-0.2448485,	delta =	0.2449977

dmin=-0.7248484848, dmax=0.7716129032

Selection Intervals for k= 2

```
s =  -2, dmin =    0.3643290, dmax =    0.7918894, delta =    0.7918894
s =  -1, dmin =    0.0733737, dmax =    0.4724301, delta =    0.1081011
s =   0, dmin =   -0.1812121, dmax =    0.1929032, delta =    0.1195295
s =   1, dmin =   -0.4058467, dmax =   -0.0537381, delta =    0.1274740
s =   2, dmin =   -0.6055219, dmax =   -0.2729749, delta =    0.1328718

dmin=-0.6055218855, dmax=0.7918894009

Selection Intervals for k=   1

s =  -2, dmin =    0.6972391, dmax =    1.3959447, delta =    1.3959447
s =  -1, dmin =    0.1314927, dmax =    0.5972965, delta =   -0.0999426
s =   0, dmin =   -0.1513805, dmax =    0.1979724, delta =    0.0664796
s =   1, dmin =   -0.3211044, dmax =   -0.0416221, delta =    0.1097584
s =   2, dmin =   -0.4342536, dmax =   -0.2013518, delta =    0.1197526

dmin=-0.4342536476, dmax=1.3959447005
```

The overlap is indicated by "delta". A set of selection rules is given next. In these rules, d and s denote the corresponding D_K and S_K, respectively.

Select S_1

```
if (d<=-0.1)                    s =  1;
else if ((d>-0.1)&(d<=0.165))   s =  0;
else                            s = -1;
```

Select S_2

```
if (d<=-0.33)                   s =  2;
```

```
        else if ((d>-0.33)&(d<=-0.1))  s =  1;
        else if ((d>-0.1)&(d<=0.1))    s =  0;
        else if ((d>0.1)&(d<=0.39))    s = -1;
        else                           s = -2;
```

Select S_3

```
        if (d<=-0.36)                  s =  2;
        else if ((d>-0.36)&(d<=-0.12)) s =  1;
        else if ((d>-0.12)&(d<=0.12))  s =  0;
        else if ((d>0.12)&(d<=0.36))   s = -1;
        else                           s = -2;
```

3. Main Recursion with Quotient Digit Prediction

Once the divisor and the dividend are transformed into the required range, we apply the following recursion on the partial remainders.

$$q_i = \lfloor R_i + \mathrm{sign}R_i * 1/2 \rfloor$$

$$R_{i+1} = 4(R_i - q_i X^*)$$

where $R_0 = Y^*$.

A direct implementation of this recursion would require three substeps:

(i) Select q_i,

(ii) Generate q_{i*X}^*, and

(iii) Compute R_{i+1}.

However, it is possible to overlap the step (i) with steps (ii) and (iii). Assume that q_1 is known. Then, define the recursion as

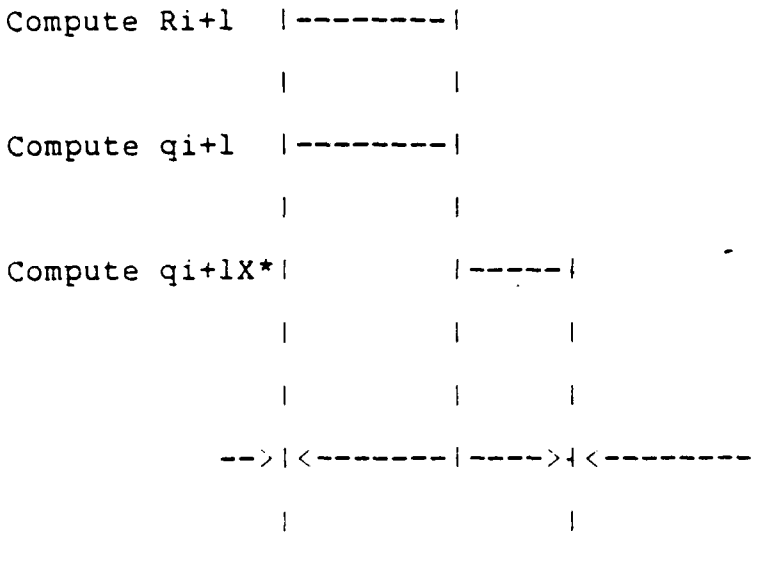
$$R_{i+1} = 4(R_i - q_i X^*)$$

$$q_{i+1} = \left\lfloor 4(\hat{R}_i - q_i + c) \right\rfloor$$

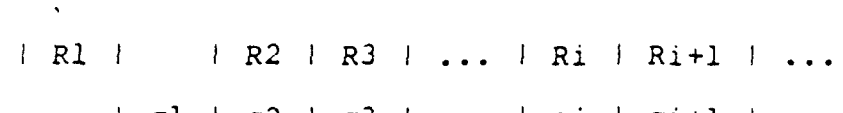
where

$$c = \begin{cases} \frac{1}{2} & \text{if } \hat{R}_i \geq q_i \\ -\frac{1}{2} & \text{otherwise} \end{cases}$$

Therefore, the recursion step contains only two substeps instead of three:



The overall timing of the main recursion would look like



4. A Complete Radix-4 Algorithm

We give a C version of the complete radix-4 division:

```
#define M  16
#define X  0.5
#define Y  0.07401786542
#define R 4
#define K 1

main()
{
        double x0, y0, d1, y1, d2, y2, d3, y3,d4;
        double quot, power;
        float r;
        double err, xprime, yprime, rem, remnext;
        int i, q, qnext, s1, s2, s3, m;

        x0 = X; y0 = Y; m = M; r = R;

/* Step 0 */

        if (x0 < 0.75 )
           {
                d1 = 2.0*x0 - 1.0;
                y1 = 2.0*y0;
           }
        else
           {
                d1 = x0 - 1.0;
                y1 = y0;
            }

/* Step 1 */

        s1 = selone(d1);
```

```
        d2 = r*d1 + s1 + s1*d1;
        y2 = y1*(1 + s1/r );

/* Step 2 */

        s2 = seltwo(d2);

        d3 = r*d2 + s2 + s2*d2/r ;
        y3 = y2*(1 + s2 / (r*r) );

/* Step 3 */

        s3 = seltre(d3);

        d4 = r*d3 + s3 + s3*d3/(r*r);
        yprime = y3*(1 + s3 / ((r*r)*r));
        xprime = d4/((r*r)*r) + 1;
        quot = 0;
        power = 1.0;
        rem = yprime;

        if (rem > 0.0 ) q = rem + 0.5;
            else q = rem - 0.5;

/* Recursion */

        for (i = 1; i < m+1 ; ++i )
                {
                 remnext = r*(rem - xprime*q);
                 qnext = select(rem, q, xprime);
                 quot = quot + q*power;
                 err = y0/x0 - quot;
                 power = power/r;
                 q = qnext; rem = remnext;
                }
}

/* Select s1 */

selone (d)
double d;
{
        int s;

        if (d <= -0.1 )                          s =  1;
        else if (( d > -0.1) & (d <= 0.165 )) s =  0;
        else                                     s = -1;

        return(s);
}

/* Select s2 */
```

```
seltwo (d)
double d;
{
        int s;

        if ( d <= -0.33 )                          s =  2;
        else if (( d > -0.33 ) & (d <= -0.1 ))  s =  1;
        else if (( d > -0.1 ) & (d <= 0.1 ))    s =  0;
        else if (( d > 0.1 ) & (d <= 0.39 ))    s = -1;
        else                                       s = -2;

        return(s);
}

/* Select s3 */

seltre (d)
double d;
{
        int s;

        if ( d <= -0.36 )                          s =  2;
        else if (( d > -0.36 ) & (d <= -0.12 )) s =  1;
        else if (( d > -0.12 ) & (d <= 0.12 ))  s =  0;
        else if (( d > 0.12 ) & (d <= 0.36 ))   s = -1;
        else                                       s = -2;

        return(s);
}

/* Select */

select (d, q, div)
double d, div;
int q;
{
        int s, k;
        double rtrunc, dtrunc;
        k = K;

/* Remainder truncated to 6 bits; divisor replaced by 1 */

        s = d * 64.0; rtrunc = s; rtrunc = rtrunc / 64.0;
        s = div * 64.0; dtrunc = s; dtrunc = dtrunc / 64.0;
        dtrunc = 1.0;

        rtrunc = ( rtrunc - q * dtrunc )* 4.0;

        if (rtrunc > 0) { s = rtrunc + 0.5;}
        else s = rtrunc - 0.5;

        return(s);
```

5. Example

```
     x0 =     0.5000000000, y0 =     0.0740178654, Q =     0.1480357308
d1 =     0.0000000000, y1 =     0.1480357308
s1 = 0

d2 =     0.0000000000, y2 =     0.1480357308
s2 = 0

d3 =     0.0000000000, y3 =     0.1480357308
s3 = 0

d4 =     0.0000000000
xprime =     1.0000000000, yprime =     0.1480357308, q1 =   0

 i        Remainder  q         Quotient              Error

 predicted next q =   1
 1     0.1480357308   0     0.0000000000     0.1480357308

 predicted next q =  -2
 2     0.592 429234   1     0.2500000000    -0.1019642692

 predicted next q =   2
 3    -1.6314283066  -2     0.1250000000     0.0230357308

 predicted next q =  -2
 4     1.4742867738   2     0.1562500000    -0.0082142692

 predicted next q =   0
 5    -2.1028529050  -2     0.1484375000    -0.0004017692

 predicted next q =  -2
 6    -0.4114116198   0     0.1484375000    -0.0004017692

 predicted next q =   1
 7    -1.6456464794  -2     0.1479492188     0.0000865121

 predicted next q =   2
 8     1.4174140826   1     0.1480102539     0.0000254769

 predicted next q =  -1
 9     1.6696563302   2     0.1480407715    -0.0000050406

 predicted next q =  -1
10    -1.3213746790  -1     0.1480369568    -0.0000012259

 predicted next q =  -1
11    -1.2854987162  -1     0.1480360031    -0.0000002723

 predicted next q =  -1
```

```
12    -1.1419948646  -1     0.1480357647    -0.0000000339

 predicted next q =   2
13    -0.5679794585  -1     0.1480357051     0.0000000258

 predicted next q =  -1
14     1.7280821658   2     0.1480357349    -0.0000000041

 predicted next q =   0
15    -1.0876713367  -1     0.1480357312    -0.0000000003

 predicted next q =  -1
16    -0.3506853469   0     0.1480357312    -0.0000000003
```

6. Binary-level Implementation

{to be done }

7. Perrormance Analysis

{to be done}

8. Alternatives

For transformation part:

- Have a small table of reciprocals of the truncated divisor, perhaps to 4-6 bits; use three stages of CSAs to multiply the divisor (2 bits per stage of the reciprocal); propagate carries to get the transformed divisor; repeat for the dividend, but do not propagate carries.

- Use radix-2 in the transformation part; possibly much simpler implementation.

- Use radix-16 in the transformation part - details worked out on the binary level; possibly fewer steps.

For recursion part:

- Implement two steps in one clock period; double the combinational logic (CSAs, selection and multiple generator).

A VLSI Design of A Radix-4 Carry Save Multiplier

M.D. Ercegovac+

UCLA Computer Science Department

University of California, Los Angeles

and

J.G. Nash

Hughes Research Laboratories,

Malibu, California

Los Angeles

April 1, 1983

Supported in part by the ONR Contract No. N00014-79-C-0866 (Research in Distributed Processing)

Authors' address: M.D. Ercegovac, 3732 Boelter Hall, UCLA, Los Angeles, CA 90024, (213)825-2660

Abstract

A scheme and a VLSI (NMOS) implementation of an area-time efficient 2-bit-at-a-time (radix-4) 2's complement multiplier are described. The scheme has a highly modular bit-slice organization and it is suitable for bus-oriented chip designs. The logic specification and the circuit design details are discussed and analyzed in terms of area-time complexity.

I. INTRODUCTION

A large class of applications, such as digital signal processing and robotics control, require extensive arithmetic capabilities. By far the most important arithmetic operation is multiplication as evidenced by the availability in the commercial market of special chips such as TRW MPY-16HJ.

There are two basic approaches to multiplication algorithms: recursive (sequential) and parallel (combinational). Parallel multipliers have higher speed and larger area requirements than the recursive multipliers.

Recursive multiplication schemes are attractive with respect to the circuit area requirements but often unacceptably slow due to their sequential mode of operation. The number of steps in the recursive algorithm is linearly proportional to the precision and the step time depends on the partial product representation and the adder type. The speed can be improved by a) recoding the multiplier into a higher radix $r=2^k$, and b) by reducing the step time using a carry-save adder. The recursion can be implemented in an obvious manner by an iterative network of carry-save adders in order to eliminate clocking overhead [HABI 70]. However, such an iterative (combinational) multiplication scheme requires approximately an n-fold area increase compared to a sequential multiplier.

AD-A141868

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA
DESIGN STUDY OF FLOATING POINT SYSTOLIC VLSI CHIP BY JG NASH, GR NUDD HUGHES RESEARCH LABORATORIES

2 OF 2
NOSC CR 232
UNCLASSIFIED
MAR 1984

One of the primary motivations for the design discussed here was the need for a fast, area-efficient multiplier that can fit well within a bus-oriented chip organization. A common criticism of the bus-oriented approach is that it is much slower than "hardwired" versions of arithmetic processors, which offer much higher speeds at reduced flexibility and programmability. We have pursued an alternative approach that combines the advantages of each. Our chip design integrates a high-speed carry-save multiplier with a conventional, slower bus structure. The design is also highly modular so that its use in other custom chips is straightforward. The bit-slice carry-save approach provides, in addition, flexibility in increasing the word lengths without large speed penalties and costly redesign.

II. THE SCHEME

The multiplicand X and the multiplier Y are n-bit fractions in 2's complement system:

$$X = (x_0, x_1, \ldots, x_{n-1})$$

$$Y = (y_0, y_1, \ldots, y_{n-1})$$

The multiplier is recoded by a triplet scanning method into the radix-4 multiplier z=y using the modified Booth's algorithm [BOOT 51, ANDE 67, RUBE 75]:

$$z = (z_0, z_1, \ldots, z_{m-1}) \quad z_i \in \{-2, -1, 0, 1, 2\}$$

where

$$m = n/2 \quad (n \text{ even})$$

$$z_j = y_{2j+1} + y_{2j+2} - 2y_{2j} \text{ for } j=0,1,\ldots,m-1$$

and $y_n = 0$

The corresponding switching expressions can be obtained from the recoding table in terms of multiplier bits y_{n-2}, y_{n-1}, and y_n :

$$M2 = y_{n-1} \oplus y_n = \begin{cases} 0 & \text{Select } 2X \\ 1 & \text{Select } X \end{cases}$$

$$M1 = y_{n-2} = \begin{cases} 0 & \text{Select direct} \\ 1 & \text{Select complement} \end{cases}$$

$$M0 = y_{n-2}y_{n-1}y_n + \bar{y}_{n-2}\bar{y}_{n-1}\bar{y}_n = \begin{cases} 0 & \text{No clear} \\ 1 & \text{Clear} \end{cases}$$

The recoder and the generator of Xz_j are organized as shown in Figure 1.

The product is obtained by the following recursion

$$P(k+1) = \frac{1}{4}(P(k) + X \cdot z_{m-1-k}) \quad k = 0,1,\ldots,m-1$$

where the initial partial product is P(0)=0. The addition operation is carried out using a carry-save adder so that the partial product P(k), for k=0 to m-1, is represented as a pair of bit-vectors (C(k), S(k)) where C is the partial carry and S is the partial sum bit-vector. The product P=XY requires assimilation of

carries using a carry-propagate adder (CPA). Since the signed operands are implicitly handled by the recoding no correction is required in the case of a negative multiplier.

The multiplication recursion is implemented as a pipeline consisting of three stages: stage S1 performs recoding, S2 generates required multiple of the multiplicand X, and S3 performs the carry-save addition. The timing of the pipeline is shown in Figure 2.

When $z_j < 0$, a negative multiple of X is formed in a standard manner by complementing the shifted/nonshifted multiplicand X and adding one in the least significant position of the adder. Since the carry-save addition operation is associative and it never causes a carry into the least significant position, the 1 required in the negation can be inserted into the LSB position after the step in which the negation was required. This can be conveniently implemented by inserting a delayed value of the M1 output of the recoder into the least significant bit of the partial carry register.

III. THE DESIGN

The multiplier was designed under assumption of a bus-oriented, bit-slice chip organization. As a result, custom VLSI chips requiring multiplier can be rapidly assembled by attaching required modules to the chip bus. The use of a carry-save, bit-

slice multiplier scheme also provides important flexibility in increasing the operand length.

The multiplier organization and its interface to the carry-propagate adder (CPA) is shown in Figure 3. It consists of a linear array of bit-slice sections, all of which are controlled by a set of circuits (MULT CONTROL in Figure 3) outside the array. In order to accommodate shifted multiplicand 2X it was necessary to append an extra full adder cell to the most significant position. In addition to the basic full adder logic, each bit-slice contains storage registers for both X and Y operands and partial sum S and carry C. The design produces only a single precision product but it can be easily extended to accomodate double precision outputs.

The interface with the CPA consists of a single set of pass transistors that connect partial sum and carries registers with the inputs to the adder.

The relation of the bit-slice sections to the controller circuit is shown in Figure 4. As mentioned above, the circuit pipeline has three stages so that three two phase clock cycles are required to fill the pipeline. During the first clock cycle, the low order multiplier bits are shifted into the storage cells, y_{n-2}, y_{n-1}, and y_n. On the ϕ_1 phase of the next clock cycle, these multiplier bits are recoded, producing M0, M1, and M2. On the ϕ_2 phase of the second clock cycle, these inputs are used to generate the corresponding multiple of the multiplicand X as dis-

cussed in Section 2. The 1-of-4 decoder performs the appropriate selection and also functions as a control line driver. The MO or clear signal overrides the select operation. The output of the select/clear function box is latched at the end of the second clock cycle. The third clock cycle consists of addition during the ϕ_1 phase followed by storage of the partial sum and carry during the ϕ_2 phase. These storage registers are initialized to zero when the multiplier is loaded with its operands.

The dual shift register, shown in Figure 5, is arranged in a fashion that allows two multiplier bits to be examined by the recoder each clock cycle. It consists of two identical shift registers, each of which holds n/2 bits. One shift register holds the odd digits, and the other holds the even digits. Each shift register is spread over two bit-slices, so that every clock cycle the data in a shift register cel advances two positions. As a result, two bits of the original radix-2 multiplier are scanned each clock.

The movement of data between the partial sum and carry storage cells and the full adder is illustrated in Figure 6. The actual addition is done in two parts, one for the carry and one for the sum. Since two multiplier bits are examined each clock cycle, it is necessary for the sum outputs of each bit-slice to be shifted to the right two bit-slices as well. It can also be seen that the carry bit is shifted to the right one-bit slice. The carry and sum logic blocks in Figure 6 each contain a storage

latch at their output.

A bit-slice layout of the multiplier is shown in Figure 7 with the pipeline stages indicated. This circuit was designed based on a NMOS depletion mode, five mask level process. All controll and I/O lines running across the slice are in metal. Note that two bus lines per slice, to support a two-port memory, are available for data transfer. Circuit area, not including the controller is approximately $12\lambda \times 2\lambda$ mil^2 where λ is the Mead-Conway scaling parameter [MEAD 80]. For example, with $\lambda = 2$, the bit-slice area is 24x4 mils^2. In Figure 8 we show an example of a chip design that uses this multiplier [NASH 82]. As can be seen the multiplier efficiently uses available space in that it takes up approximately the same amount of area as the CPA. The control circuitry is not shown.

IV. DISCUSSION

In this section we estimate the area-time efficiency of this multiplier. The maximum operating speed of the multiplier is detremined by the slowest stage in the pipeline. There are six phases of activity in the three-stage pipeline as listed in Table I. The delays in all sections of the pipeline are determined by approximately two gating levels per half clock cycle. From Figure 4 it is clear that the largest drive requirements are placed upon the 1-of-4 decoder, which must charge the select/clear lines across all bits. However, because the controller circuits are

outside the bit-slice array, there is space enough to make them as large as desired. As a result, the recoder section can also be used as an intermediate driver stage for the 1-of-4 decoder/driver. This arrangement provides a capability to activate the select/clear blocks with delays comparable to the delay through the full adder and the shift register. Simulations of the multiplier stages, including layout parasitics, indicated that for 6μ gate lengths, the pipeline slowest stage of 33μs was in the full adder bit-slice section. Thus, the clock speed for the multiplier is expected to be about 16MHz.

The area-time complexity of the multiplier (apart from the CPA) can then be expressed as:

$$AT_{(r=4)} = 3nA_{FA} \times (\frac{n}{2}+3)t_{FA}$$

where A_{FA} is the full adder area and t_{FA} is the propagation time through the full adder. Here, we have used as an estimate for the bit-slice area, three times the full adder area in the bit-slice. Assuming 6μ gate lengths, a 32-bit fixed-point multiplier would require about 500ns. Using a 3μ technology we estimate that about 200-250ns would be required to perform this multiplication.

The corresponding area-time for a radix-2 (no recoding) iterative or combinational (array) multiplier, also excluding the CPA, is approximately

$$AT_{(r=2)} = n(n-1)A_{FA} \times nt_{FA}.$$

For large n, the radix-4 recursive multiplier approach provides a

factor of n/3 improvement in the area-time product with respect to the combinational array multiplier.

Acknowledgement

The authors are indebted to G.R. Nudd of HRL for his comments and support.

References

[ANDE 67] Anderson, S.F. et al., "The IBM System 360/ Model 91: Floating-Point Execution Unit", IBM J.Res.Develop., January 1967, pp.35-53.

[BOOT 51] Booth,A.D., "A Signed Binary Multiplication Technique", Quart.J.Mech.Appl.Math., Vol.4, part 2, 1951.

[HABI 70] Habibi,A. and Wintz, P.A., "Fast Multipliers", IEEE Trans. on Computers, Vol.C-19, February 1970, pp.153-157.

[MEAD 80] Mead, C. and L. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980.

[NASH 81] Nash, J.G., Nudd, G.R., and Hansen, S. "Concurrent Architectures for Toeplitz Linear System Solution", Proc. Govt. Microcircuit Appl. Conf., Orlando, Fla., November 2, 1982.

[RUBI 75] Rubinfield, L.P., "A Proof of the Modified Booth's Algorithm for Multiplication", IEEE Trans. on Computers, Vol.C-24, October 1975, pp.1014-1015.

FIGURE CAPTIONS

Figure 1: Recoder and multiple generator.

Figure 2: Timing Diagram

Figure 3: Bit-slice arrangement of multiplier and interface to carry propagate adder (CPA).

Figure 4: Schematic of multiplier pipeline showing controller circuits (outside array of bit-slice elements) and bit-slice functional arrangement.

Figure 5: Diagram of dual shift register. A storage register, not shown is used to store the multiplicand X.

Figure 6: Schematic of data flow in partial product carry-save add/shift section of serial multiplier.

Figure 7: Layout of bit-slice section of multiplier using NMOS, depletion load, five mask level process.

Figure 8: (a) Chip organization and (b) micorphotograph of bus oriented arithmetic processing chip incorporating carry-save multiplier. This chip is 28-bits, fixed point made using 6u NMOS technology. Chip clock of 2-4 MHz is synchronized with multiplier 16 MHz clock.

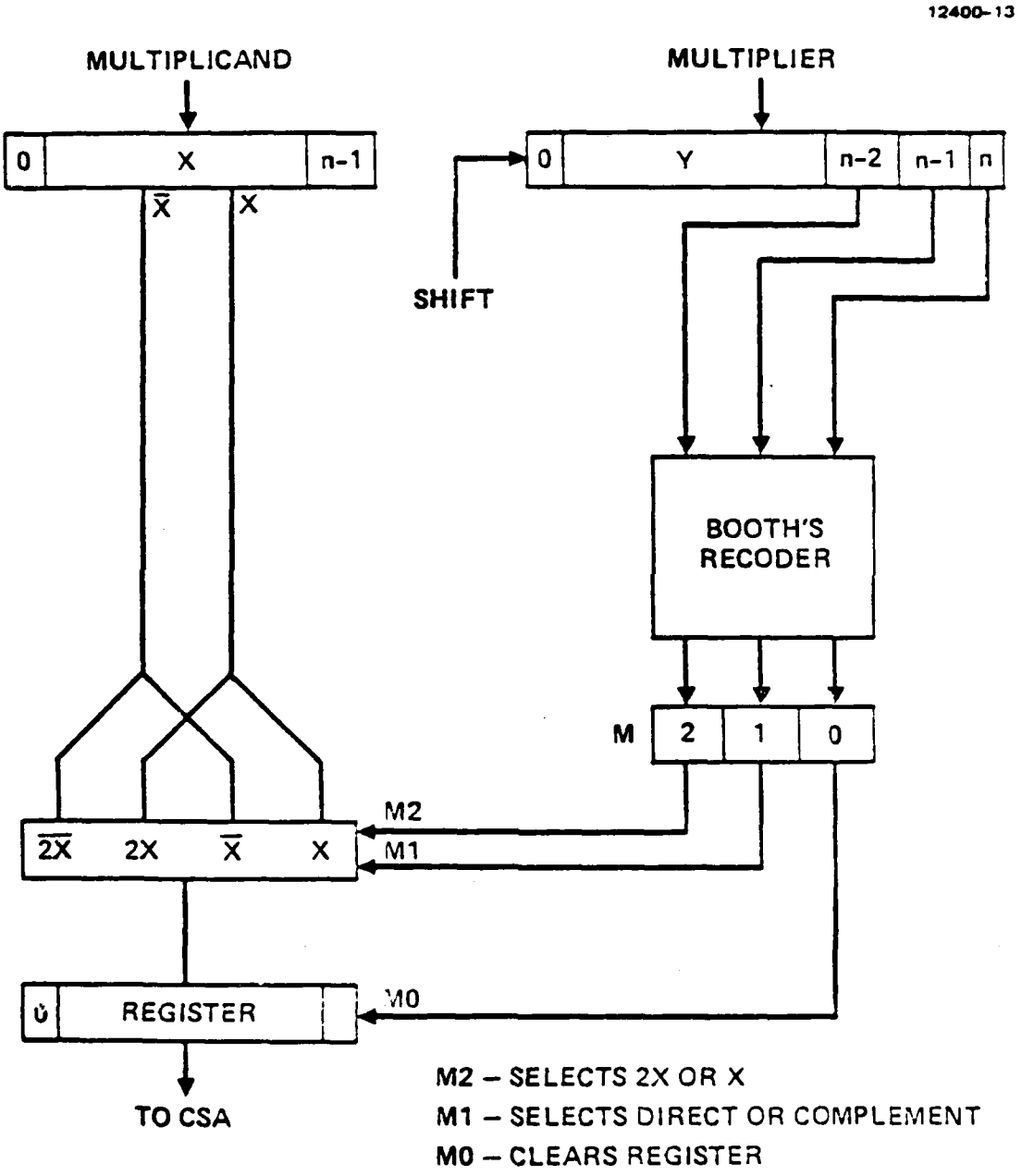

12400-13
MULTIPLICAND
MULTIPLIER
0
X
n-1
0
Y
n-2
n-1
n
$\overline{X}$
X
SHIFT
BOOTH'S RECODER
M
2
1
0
M2
M1
$\overline{2X}$
2X
$\overline{X}$
X
0
REGISTER
M0
TO CSA
M2 – SELECTS 2X OR X
M1 – SELECTS DIRECT OR COMPLEMENT
M0 – CLEARS REGISTER

FIGURE 1

	0	1	2	3	4	. . .	m+1	m+2	(CPA)
	Load X								
	Load Y								
S1	0	z_{m-1}	z_{m-2}	z_{m-3}	z_{m-4}	. . .	-	-	
S2	0	0	A_{m-1}	A_{m-2}	A_{m-3}	. . .	A_0	-	
S3	0	0	0	P(1)	P(2)	. . .	P(m-1)	P(m)	

Figure 2: Timing diagram of the multiplication pipeline

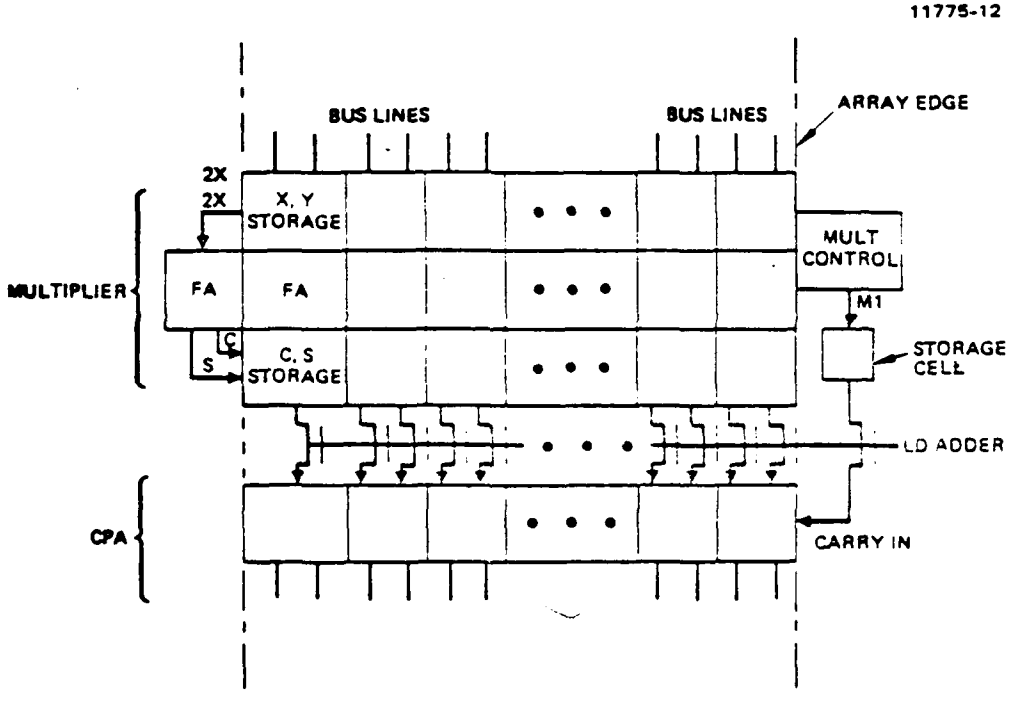

11775-12
BUS LINES
BUS LINES
ARRAY EDGE
2X
2X
X, Y
STORAGE
MULT
CONTROL
MULTIPLIER
FA
FA
M1
C
S
C, S
STORAGE
STORAGE
CELL
LD ADDER
CPA
CARRY IN

FIGURE 3

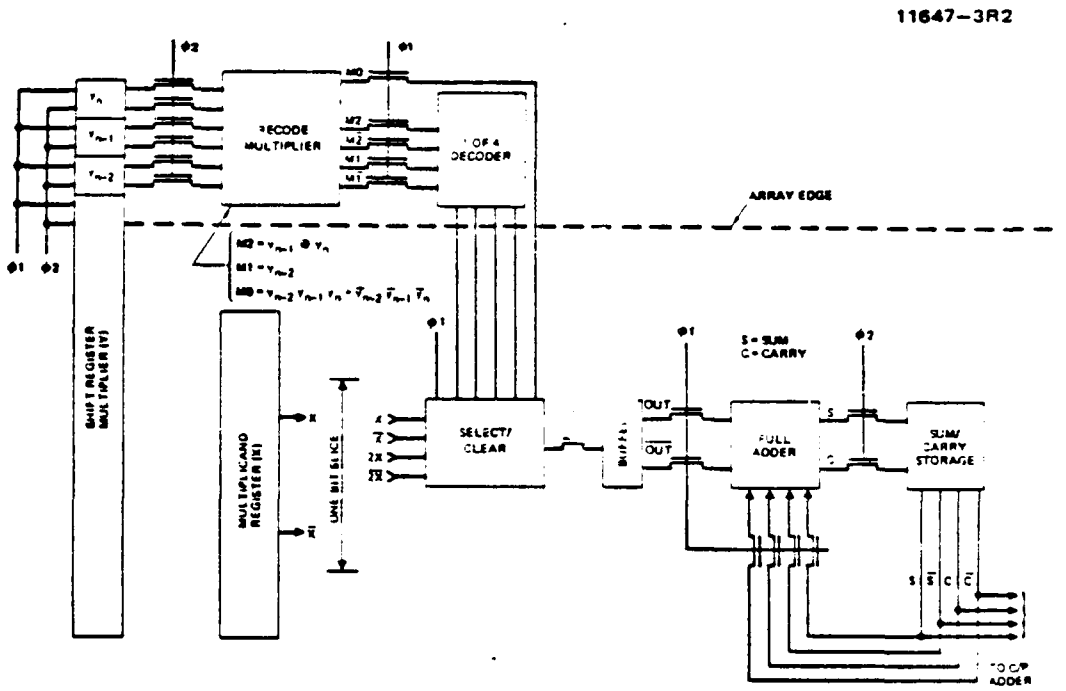

11647-3R2
ARRAY EDGE
RECODE MULTIPLIER
1 OF 4 DECODER
SHIFT REGISTER MULTIPLIER (Y)
MULTIPLICAND REGISTER (X)
ONE BIT SLICE
SELECT/ CLEAR
BUFFER
FULL ADDER
SUM/ CARRY STORAGE
S = SUM
C = CARRY
TO C/P ADDER

FIGURE 4

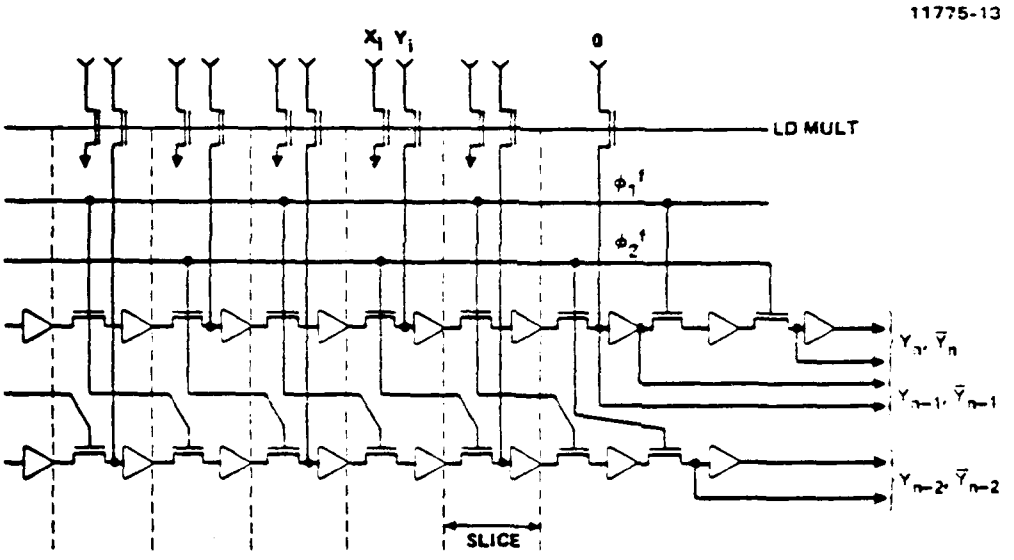

11775-13
X_i Y_i
0
LD MULT
$\phi_1{}^t$
$\phi_2{}^t$
Y_n, $\overline{Y}_n$
Y_{n-1}, $\overline{Y}_{n-1}$
Y_{n-2}, $\overline{Y}_{n-2}$
SLICE

FIGURE 5

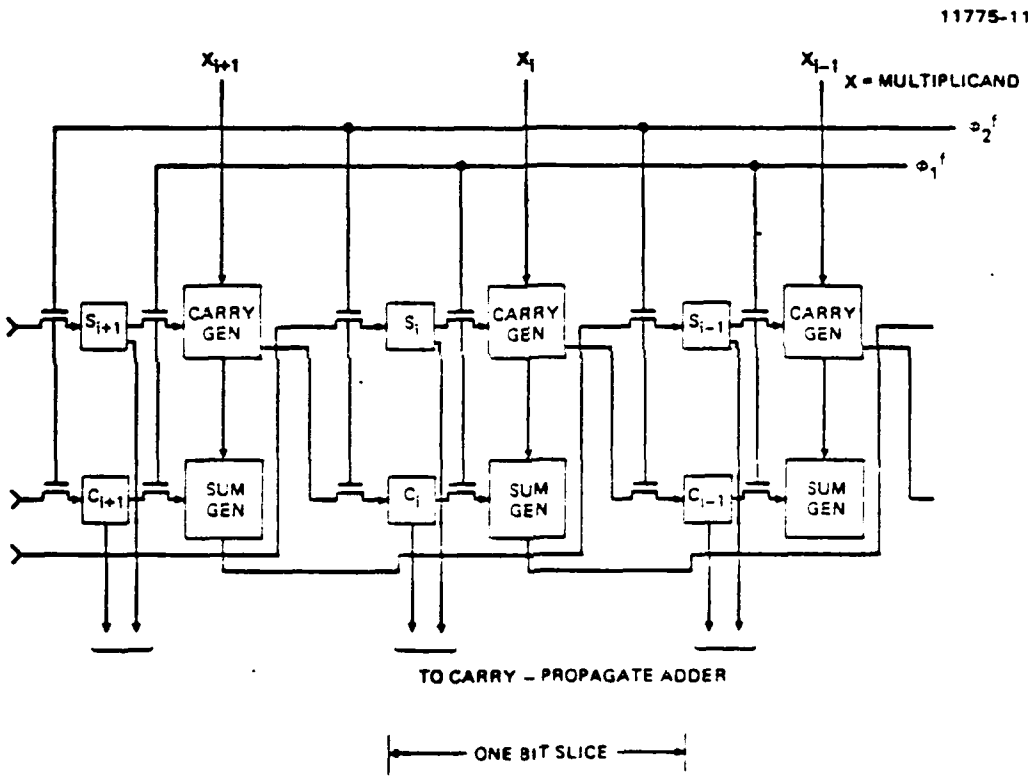

11775-11
X_{i+1}
X_i
X_{i-1}
X = MULTIPLICAND
Φ_2
Φ_1
S_{i+1}
CARRY GEN
S_i
CARRY GEN
S_{i-1}
CARRY GEN
C_{i+1}
SUM GEN
C_i
SUM GEN
C_{i-1}
SUM GEN
TO CARRY – PROPAGATE ADDER
ONE BIT SLICE

FIGURE 6

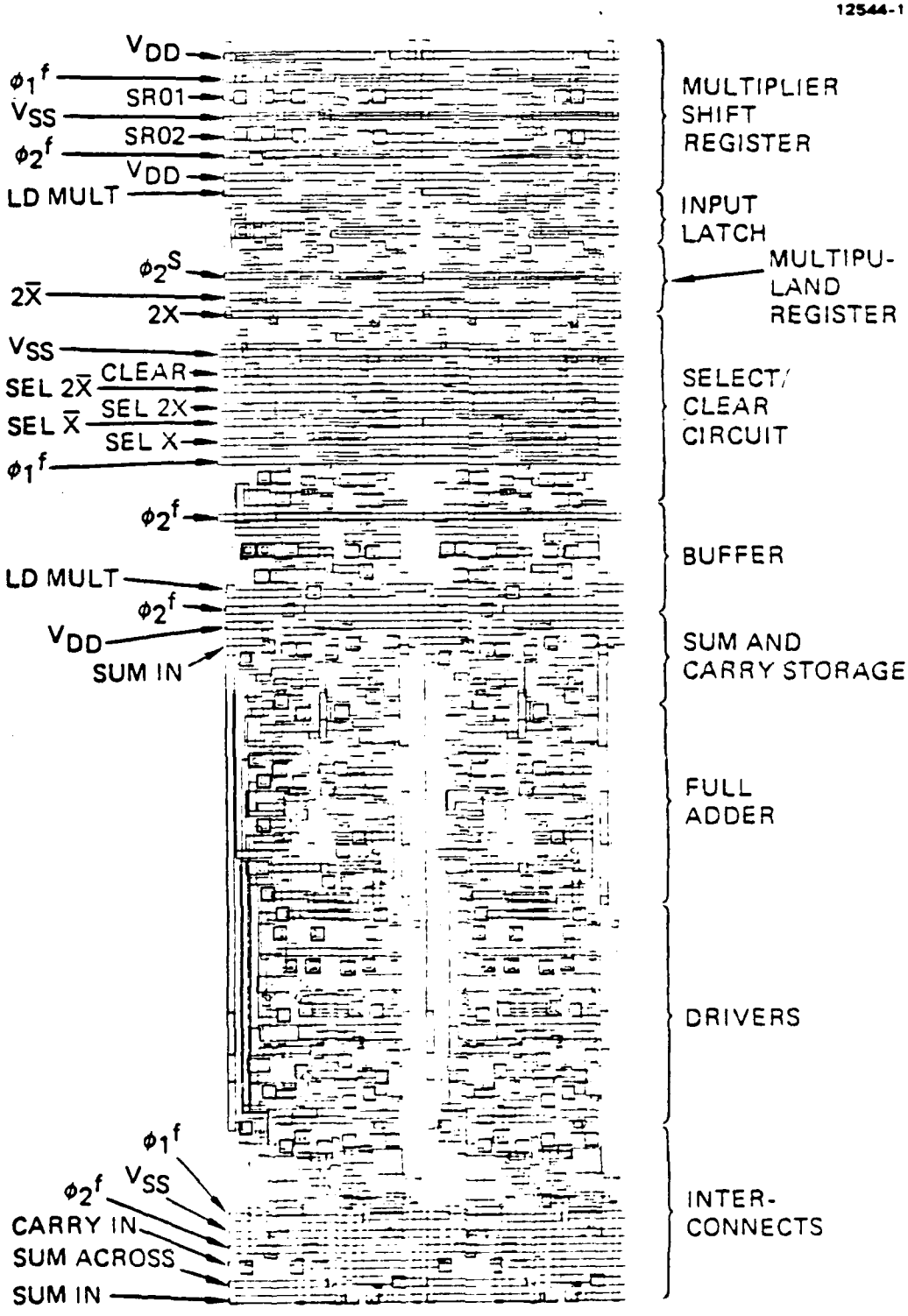
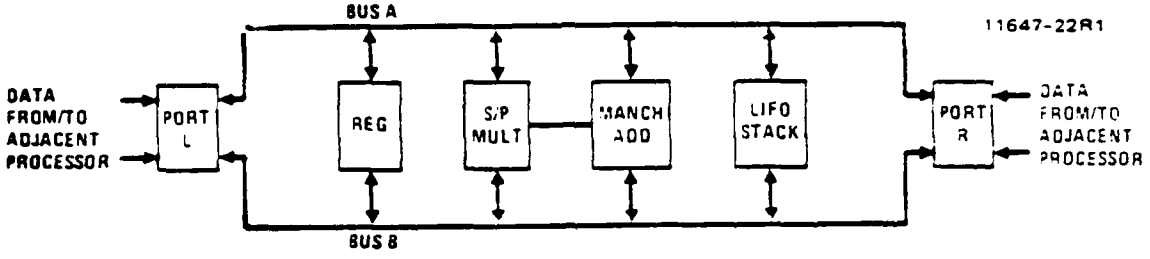

12544-1
V_{DD}
$\phi_1{}^f$
SR01
V_{SS}
SR02
$\phi_2{}^f$
V_{DD}
LD MULT
$\phi_2{}^S$
$2\bar{X}$
2X
V_{SS}
SEL $2\bar{X}$
CLEAR
SEL 2X
SEL $\bar{X}$
SEL X
$\phi_1{}^f$
$\phi_2{}^f$
LD MULT
$\phi_2{}^f$
V_{DD}
SUM IN
$\phi_1{}^f$
$\phi_2{}^f$
V_{SS}
CARRY IN
SUM ACROSS
SUM IN
MULTIPLIER SHIFT REGISTER
INPUT LATCH
MULTIPU-LAND REGISTER
SELECT/ CLEAR CIRCUIT
BUFFER
SUM AND CARRY STORAGE
FULL ADDER
DRIVERS
INTER-CONNECTS

FIGURE 7


BUS A
11647-22R1
DATA FROM/TO ADJACENT PROCESSOR
PORT L
REG
S/P MULT
MANCH ADD
LIFO STACK
PORT R
DATA FROM/TO ADJACENT PROCESSOR
BUS B

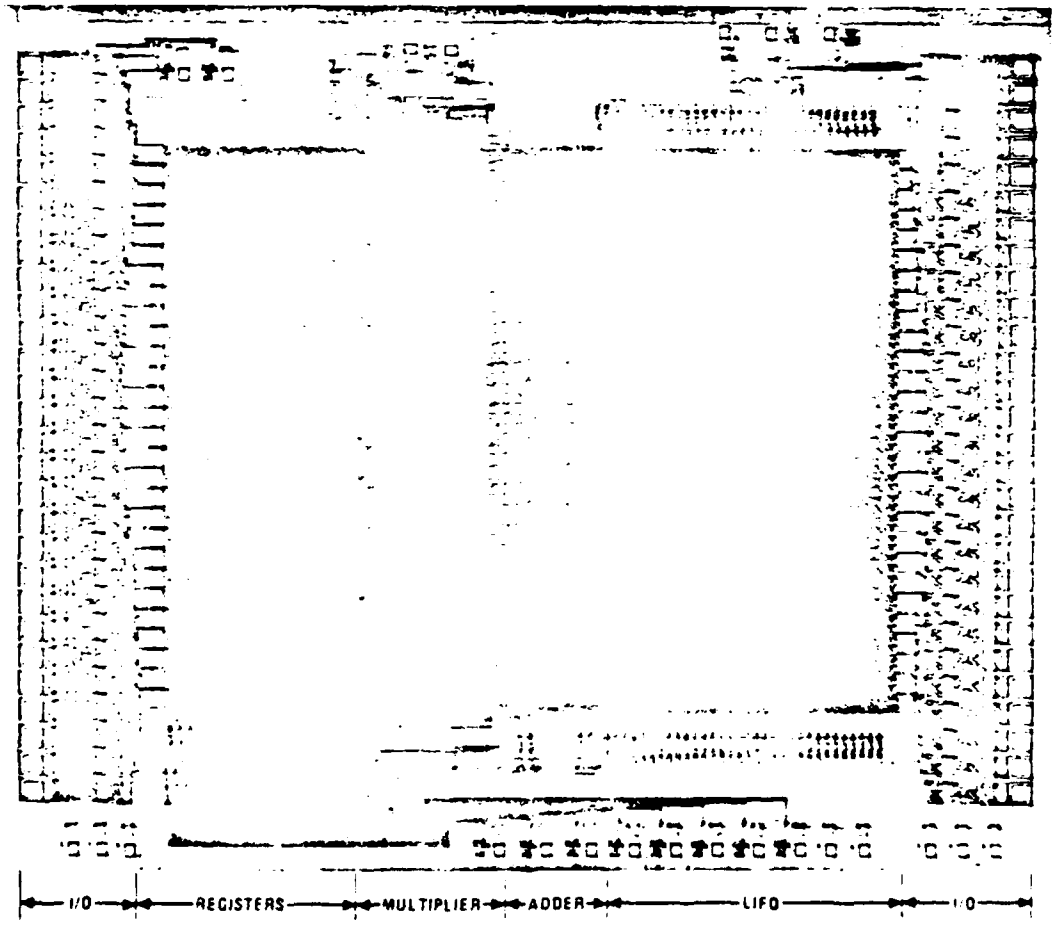

I/O
REGISTERS
MULTIPLIER
ADDER
LIFO
I/O

FIGURE A

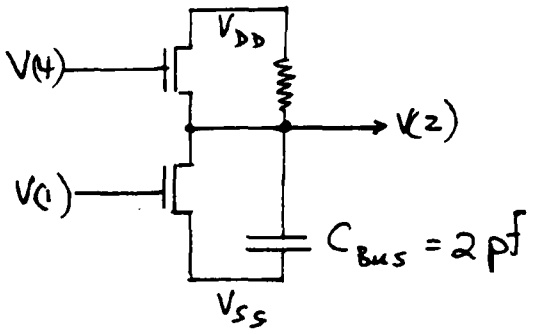
TABLE I

Clock Phase	Pipeline Activity
1	Shift Register-One Bit Slice Shift
2	Shift Register-One Bit Slice Shift
1	Recode Multiplier Bits
2	1 of 4 Nand Decoder
1	Precharge Select/Clear Circuits
2	Drive Select/Clear Lines
1	Partial Product Addition
2	Partial Product Storage

APPENDIX C

TRANSIENT RESPONSE

TIME OF PROCESSOR BUS



MOSFET

MODEL	MPULLDOWN ENHAN	MPULLUP ENHAN
[illegible]	1.93D-[illegible]	1.93D-[illegible]
[illegible]	0.0	-4.999
[illegible]	4.999	0.001
[illegible]	0.0	-4.999

SPICE 2E.1 (16APR76) -- TRANSIENT ANALYSIS

TIME	V(2)	V(1)	V(4)	I(VDD)
[illegible]	5.00D+00	0.0	0.0	-2.08D-11
1.000D-09	5.00D+00	0.0	0.0	-2.08D-11
1.000D-09	4.02D+00	5.00D+00	0.0	-9.76D-09
1.500D-08	2.89D+00	5.00D+00	0.0	-2.12D-08
2.000D-08	1.90D+00	5.00D+00	0.0	-3.10D-08
2.500D-08	1.17D+00	5.00D+00	0.0	-3.83D-08
3.000D-08	6.76D-01	5.00D+00	0.0	-4.32D-08
3.500D-08	3.78D-01	5.00D+00	0.0	-4.62D-08
4.000D-08	2.07D-01	5.00D+00	0.0	-4.79D-08
4.500D-08	1.12D-01	5.00D+00	0.0	-4.89D-08
5.000D-08	5.99D-02	5.00D+00	0.0	-4.94D-08
5.500D-08	3.21D-02	5.00D+00	0.0	-4.97D-08
6.000D-08	1.72D-02	5.00D+00	0.0	-4.98D-08
6.500D-08	9.25D-03	5.00D+00	0.0	-4.99D-08
7.000D-08	5.02D-03	5.00D+00	0.0	-5.00D-08
7.500D-08	2.76D-03	5.00D+00	0.0	-5.00D-08
8.000D-08	1.56D-03	5.00D+00	0.0	-5.00D-08
8.500D-08	9.22D-04	5.00D+00	0.0	-5.00D-08
9.000D-08	5.82D-04	5.00D+00	0.0	-5.00D-08
9.500D-08	4.02D-04	5.00D+00	0.0	-5.00D-08
1.000D-07	3.05D-04	5.00D+00	0.0	-5.00D-08
1.050D-07	2.54D-04	5.00D+00	0.0	-5.00D-08
1.100D-07	-9.51D-02	0.0	0.0	-5.10D-08
1.150D-07	-9.50D-02	0.0	0.0	-5.10D-08
1.200D-07	-9.49D-02	0.0	0.0	-5.10D-08
1.250D-07	8.12D-01	0.0	5.00D+00	-3.15D-04
1.300D-07	1.44D+00	0.0	5.00D+00	-2.09D-04
1.350D-07	1.88D+00	0.0	5.00D+00	-1.48D-04
1.400D-07	2.19D+00	0.0	5.00D+00	-1.10D-04
1.450D-07	2.42D+00	0.0	5.00D+00	-8.42D-05
1.500D-07	2.60D+00	0.0	5.00D+00	-6.63D-05
1.550D-07	2.74D+00	0.0	5.00D+00	-5.33D-05
1.600D-07	2.86D+00	0.0	5.00D+00	-4.37D-05
1.650D-07	2.96D+00	0.0	5.00D+00	-3.63D-05
1.700D-07	3.04D+00	0.0	5.00D+00	-3.06D-05
1.750D-07	3.11D+00	0.0	5.00D+00	-2.61D-05
1.800D-07	3.16D+00	0.0	5.00D+00	-2.24D-05
1.850D-07	3.22D+00	0.0	5.00D+00	-1.95D-05
1.900D-07	3.26D+00	0.0	5.00D+00	-1.70D-05
1.950D-07	3.30D+00	0.0	5.00D+00	-1.50D-05
2.000D-07	3.33D+00	0.0	5.00D+00	-1.33D-05

SUMMARY OF ERRORS FOR THIS JOB ERROR NUMBER NUMBER OF ERROR

308

30nsec BUS DISCHARGE TIME THROUGH $\frac{W}{L}=7$ PASS TRANSISTOR

~80 NSEC BUS CHARGE TIME (TO 3.5 V)

...SE DISCHARGE RATE (BUS)

...ING TEMPERATURE = ...000 DEG ...

...ENHAN ... = 5.99999E-01 L= 4.99999E-04
...ENHAN W= 5.99999E-01 L= 4.99999E-04

BUS CAPACITANCE

...PULSE 0 5 5NSEC 1NSEC 1NSEC 100NSEC 500NSEC)
...PULSE 0 5 10NSEC 1NSEC 1NSEC 100NSEC 500NSEC)
TRAN 5NSEC 200NSEC
...PRINT TRAN ... (VDD)
MODEL FOR ENHANCEMENT AND DEPLETION MODE TRANSISTORS
...DEL ENHAN NMOS(LEVEL=2 VTO= 5.99999E-01 KP=11E-6 GAMMA=0.38 RD=...
...CGS=3.8E-12 CGD=3.8E-12 CGP=0.00 CBD=1.6E-8 CBS=1.6E-8
...00E-8 PB=0.7 XJ=1.1E-4 LD=0.7 NGATE=1.0 TPS=+1
...2 A2=0.6 UCRIT=2E4 UEXP=0.3 UTRA=0)
...DEL DEPL NMOS(LEVEL=2 VTO=-3.50000E+00 KP=11E-6 GAMMA=0.38 RD=...
-RS=100 CGS=3.8E-12 CGD=3.8E-12 CGB=0.00 CBD=1.6E-8 CBS=1.6E-8
-OX=500E-8 PB=0.7 XJ=1.1E-4 LD=0.7 NGATE=1.0 TPS=+1
-A1=0.2 A2=0.6 UCRIT=2E4 UEXP=0.3 UTRA=0)
OPTION NOMOD NOPAGE NUMDGT=3
END
...SPICE 2E.1 (16APR78) -- INITIAL TRANSIENT SOLUTION

NODE	VOLTAGE	NODE	VOLTAGE	NODE	VOLTAGE	NODE	VOLTAG
...	...	2	4.9998...	4	0.0	5	5.

...AGE SOURCE CURRENTS

NAME	CURRENT
...D	...980D-...
...P	0.0
...C	...

END
DATE
FILMED

MAY 7, 1984